

国外计算机科学教材系列

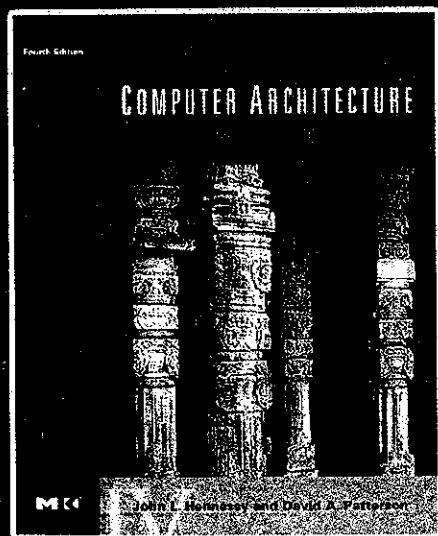


权威作者 经典教材

计算机系统结构

—— 量化研究方法 (第四版)

Computer Architecture:
A Quantitative Approach, Fourth Edition



[美] John L. Hennessy 著
David A. Patterson

白跃彬 译 钱德沛 审校



电子工业出版社

Publishing House of Electronics Industry
<http://www.phei.com.cn>

国外计算机科学教材系列



计算机系统结构 ——量化研究方法

(第四版)

Computer Architecture: A Quantitative Approach
Fourth Edition

[美] John L. Hennessy 著
David A. Patterson

白跃彬 译
钱德沛 审校

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

本书堪称计算机系统结构学科的“圣经”，是计算机体系结构方向的学生的必读教材。全书系统地介绍了计算机系统的设计基础、指令集系统结构、流水线与指令级并行技术、层次化存储系统与存储设备、互连网络以及多处理器系统等重要内容。在这一最新版本中，作者更新了从单核处理器到多核处理器的历史发展过程的相关内容，同时使用了广受好评的“量化研究方法”进行计算设计，并阐述了多种可以实现并行的技术，这些技术恰恰是展现多处理器系统结构威力的关键。在介绍多处理器时，作者不仅讲述了处理器的性能，而且还介绍了处理器性能之外的其他设计要素，包括功耗、可靠性、可用性和可信性等。

本书可作为计算机专业计算机系统结构方向的高年级本科生及研究生的教材，也可以作为相关技术人员的参考书。

Authorized translation from the English language edition published by Elsevier Science(USA). Copyright © 2007 by Elsevier Science(USA).

Translation Copyright © 2007 by Publishing House of Electronics Industry.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

本书中文简体专有翻译出版权由 Elsevier Science(USA)授予电子工业出版社。其原文版权及中文翻译出版权受法律保护。未经许可，不得以任何形式或手段复制或抄袭本书内容。

版权贸易合同登记号 图字：01-2006-7674

图书在版编目 (CIP) 数据

计算机系统结构：量化研究方法：第4版 / (美)亨尼西 (Hennessy, J. L.) 等著；白跃彬译. - 北京：电子工业出版社，2007.8

(国外计算机科学教材系列)

书名原文：Computer Architecture: A Quantitative Approach, Fourth Edition

ISBN 978-7-121-04769-5

I. 计... II. ①亨... ②白... III. 计算机体系结构-教材 IV. TP303

中国版本图书馆 CIP 数据核字 (2007) 第 114263 号

责任编辑：谭海平

印 刷：北京市海淀区四季青印刷厂

装 订：涿州市桃园装订有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787 × 1092 1/16 印张：32.5 字数：915 千字

印 次：2009 年 6 月第 2 次印刷

定 价：69.80 元 (附光盘 1 张)

凡所购买电子工业出版社的图书有缺损问题，请向购买书店调换；若书店售缺，请与本社发行部联系。联系及邮购电话：(010) 88254888。

质量投诉请发邮件至 zlt@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。

出版说明

21 世纪初的 5 至 10 年是我国国民经济和社会发展的关键时期,也是信息产业快速发展的关键时期。在我国加入 WTO 后的今天,培养一支适应国际化竞争的一流 IT 人才队伍是我国高等教育的重要任务之一。信息科学和技术方面人才的优劣与多寡,是我国面对国际竞争时成败的关键因素。

当前,正值我国高等教育特别是信息科学领域的教育调整、变革的重大时期,为使我国教育体制与国际化接轨,有条件的高等院校正在为某些信息学科和技术课程使用国外优秀教材和优秀原版教材,以使我国在计算机教学上尽快赶上国际先进水平。

电子工业出版社秉承多年来引进国外优秀图书的经验,翻译出版了“国外计算机科学教材系列”丛书,这套教材覆盖学科范围广、领域宽、层次多,既有本科专业课程教材,也有研究生课程教材,以适应不同院系、不同专业、不同层次的师生对教材的需求,广大师生可自由选择和自由组合使用。这些教材涉及的学科方向包括网络与通信、操作系统、计算机组织与结构、算法与数据结构、数据库与信息处理、编程语言、图形图像与多媒体、软件工程等。同时,我们也适当引进了一些优秀英文原版教材,本着翻译版本和英文原版并重的原则,对重点图书既提供英文原版又提供相应的翻译版本。

在图书选题上,我们大都选择国外著名出版公司出版的高校教材,如 Pearson Education 培生教育出版集团、麦格劳-希尔教育出版集团、麻省理工学院出版社、剑桥大学出版社等。撰写教材的许多作者都是蜚声世界的教授、学者,如道格拉斯·科默(Douglas E. Comer)、威廉·斯托林斯(William Stallings)、哈维·戴特尔(Harvey M. Deitel)、尤利斯·布莱克(Uyless Black)等。

为确保教材的选题质量和翻译质量,我们约请了清华大学、北京大学、北京航空航天大学、复旦大学、上海交通大学、南京大学、浙江大学、哈尔滨工业大学、华中科技大学、西安交通大学、国防科学技术大学、解放军理工大学等著名高校的教授和骨干教师参与了本系列教材的选题、翻译和审校工作。他们中既有讲授同类教材的骨干教师、博士,也有积累了几十年教学经验的老教授和博士生导师。

在该系列教材的选题、翻译和编辑加工过程中,为提高教材质量,我们做了大量细致的工作,包括对所选教材进行全面论证;选择编辑时力求达到专业对口;对排版、印制质量进行严格把关。对于英文教材中出现的错误,我们通过与作者联络和网上下载勘误表等方式,逐一进行了修订。

此外,我们还将与国外著名出版公司合作,提供一些教材的教学支持资料,希望能为授课老师提供帮助。今后,我们将继续加强与各高校教师的密切联系,为广大师生引进更多的国外优秀教材和参考书,为我国计算机科学教学体系与国际教学体系的接轨做出努力。

电子工业出版社

教材出版委员会

- | | | |
|----|-----|---|
| 主任 | 杨芙清 | 北京大学教授
中国科学院院士
北京大学信息与工程学部主任
北京大学软件工程研究所所长 |
| 委员 | 王 珊 | 中国人民大学信息学院院长、教授 |
| | 胡道元 | 清华大学计算机科学与技术系教授
国际信息处理联合会通信系统中国代表 |
| | 钟玉琢 | 清华大学计算机科学与技术系教授、博士生导师
清华大学深圳研究生院信息学部主任 |
| | 谢希仁 | 中国人民解放军理工大学教授
全军网络技术研究中心主任、博士生导师 |
| | 尤晋元 | 上海交通大学计算机科学与工程系教授
上海分布计算技术中心主任 |
| | 施伯乐 | 上海国际数据库研究中心主任、复旦大学教授
中国计算机学会常务理事、上海市计算机学会理事长 |
| | 邹 鹏 | 国防科学技术大学计算机学院教授、博士生导师
教育部计算机基础课程教学指导委员会副主任委员 |
| | 张昆藏 | 青岛大学信息工程学院教授 |

多处理器时代的到来已经势不可挡。当我们告别单核处理器时代进入芯片多处理时代之际，Hennessy和Patterson著作最新版本的推出无疑是逢时之作。很少有其他著作能像本书一样对计算机系统结构产生过如此巨大的影响，而这一最新版本必定在未来很长一段时间内是这一领域的权威之作。

—— Luiz André Barroso, Google 公司

如果你问下列哪个可算做经典：甲壳虫乐队，HP 计算器，巧克力曲奇饼，还是这本书？我会告诉你，它们都很经典，因为它们都经受住了时间的考验！

—— Robert P. Colwell, Intel 公司首席架构师

这本书不仅提供了一个应当被计算机系统架构师了熟于心的权威参考，同时也对业界的趋势进行了积极探索。

—— Krisztián Flautner, ARM 公司

青出于蓝而胜于蓝！新版的内容更新后，与当今计算机系统结构领域的核心问题更加密切相关。而且，其中新的练习题对学生和老师都更加实用了。

—— Norman P. Jouppi, HP 实验室

本书以导致 RISC 技术革命的基本原理为基础，其中包括导致 CISC 转换的因素。现在，在这个新版本中，它清晰地解释并阐述了最近出现的新一代多线程多核处理器所需的微系统结构技术。

—— Marc Tremblay, Fellow & VP, 首席架构师, Sun 微系统公司

这本著作具备了优秀教科书的所有优点：它介绍的计算机组织和设计的观点和技术，符合教学法，易于接受；它的内容吸引人；同时还涉及到非常多的领域。该书的第一版堪称内容精良的典范，而最新版又再次做到了这一点。

—— Milos Ercegovac, 加州大学洛杉矶分校

他们再次做到了这一点。Hennessy 和 Patterson 有力地证明了为什么他们会成为这个博大精深却又日新月异的领域中的泰斗。谬误：计算机系统结构在信息时代并非必不可少。陷阱：你不需要本书。

—— Michael D. Smith, 哈佛大学

Hennessy 和 Patterson 再次做到了这一点！第四版是一次经典重现，通过调整，它很好地应对了由所谓‘后 CMOS 时代’技术不断变化而带来的一些制约条件。书中对真实产品所做的详细的

案例分析对教学非常有益,而且其行文流畅让人爱不释手。这本书对学生和专业人士等人来说不容错过。

—— Pradip Bose, IBM 公司

本书最新版向学生们讲述了系统结构的框架和基础知识,这些都是未来的优秀架构师所必需的。

—— Ravishankar Iyer, Intel 公司

随着技术的进步,设计方面机遇与约束条件的变化,使得这本书也需要与时俱进。第四版秉承以前的风格,在呈现了商业产品革新的同时描述了以下一些基本的概念:高级处理器和存储器系统设计技术,多线程和芯片多处理器,存储系统,虚拟机,等等。该书对于那些想要学习实际商业产品所用的系统结构概念的人来说是绝佳的资源。

—— Gurindar Sohi, 威斯康星大学麦迪逊分校

非常高兴能够让我的学生使用这本好书来学习计算机系统结构方面的知识,不过,我也因为自己没有能写出这样的好书而羡慕该书的作者。

—— Mateo Valero, UPC, 巴塞罗那

Hennessy和Patterson继续根据当前计算机系统设计领域的变化对他们的教学方式进行了改进。学生们由此得以深入理解影响计算机系统结构设计的因素以及计算机系统领域潜在的研究方向。

—— Dan Connors, 科罗拉多大学布德尔分校

经过再版,本书仍将是以后计算机系统结构领域学生们的必读书目。

—— Wen-mei Hwu, 伊利诺伊大学香槟分校

第四版秉承了其一贯采用的内容精挑细选、方法新颖的风格,这非常受学生、研究人员以及计算机系统设计者们的欢迎。新版中的课程对读者将一直都会有很大的帮助。

—— David Brooks, 哈佛大学

在第四版中, Hennessy 和 Patterson 使得本书回归到突出个别重点的做法,因而第四版再次像第一版一样经典。

—— Mark D. Hill, 威斯康星大学麦迪逊分校

计算机系统结构相关公式

1. CPU 时间 = 指令数 × 每条指令的时钟周期数 × 时钟周期所占时间

2. Amdahl 定律 (阿姆达尔定律):

$$\text{总加速比} = \frac{\text{原执行时间}}{\text{改进后执行时间}} = \frac{1}{(1 - \text{可改进任务占任务总量的百分比}) + \frac{\text{可改进任务占任务总量的百分比}}{\text{改进部分加速比}}}$$

3. 动态功率 = $1/2 \times \text{电容性负载} \times \text{电压的平方} \times \text{交换频率}$

4. 静态功率 = 静态电流 × 电压

5. 平均存储器访问时间 = 命中时间 + 缺失率 × 缺失代价

6. 有效性 = 平均失败时间 / (平均失败时间 + 平均修复时间)

7. 晶片产量 = 晶片成品率 × $\left(1 + \frac{\text{单位面积缺陷} \times \text{晶片面积}}{\alpha}\right)^{-\alpha}$

式中, 晶片成品率表示因已经报废而无须测试的晶片数, α 表示掩膜层数, 是影响到晶片产量的重要参数 (通常 $\alpha = 4.0$)

8. 每条指令缺失数 = 缺失率 × 每条指令存储器访问数

9. Cache 索引空间: $2^{\text{index}} = \text{Cache 容量} / (\text{块大小} \times \text{组相关性})$

10. 算数平均 (AM)、加权平均 (WAM) 和几何平均 (GM):

$$\text{AM} = \frac{1}{n} \sum_{i=1}^n \text{Time}_i \quad \text{WAM} = \sum_{i=1}^n \text{Weight}_i \times \text{Time}_i \quad \text{GM} = \sqrt[n]{\prod_{i=1}^n \text{Time}_i} = \exp\left(\frac{1}{n} \times \sum_{i=1}^n \ln(\text{Time}_i)\right)$$

式中, Time_i 是 n 个程序中第 i 个的执行时间, Weight_i 是第 i 个程序的权重。

$$11. \text{几何标准偏差} = \exp\left(\sqrt{\frac{\sum_{i=1}^n \ln(\text{Time}_i) - \ln(\text{几何平均})^2}{n}}\right)$$

经验规律

1. Amdahl/Case 定律: 要使计算机系统较为平衡, 每 MIPS 的 CPU 性能需要对应 1 MB 的主存容量和 1 Mb/s 的 I/O 带宽。

2. 90/10 局部性原理: 一个程序中 10% 的代码执行其 90% 的指令。

3. 带宽定律: 带宽的增长速度至少是其时延性能提高速度的两倍。

4. 2:1 Cache 定律: 大小为 N 的直接映射 Cache 的缺失率大致和大小为 $N/2$ 的双路组相连 Cache 缺失率相同。

5. 可靠性定律: 进行毫无错误的设计。

MIPS 64 常用指令子集

指令类型 / 操作码	指令含义
数据传输指令	在寄存器和存储器间,或在整型和浮点型或专用寄存器间传输数据;只有存储器地址模式是16位位移量+通用寄存器的值
LB, LBU, SB	装载一个字节(到整型寄存器),装载一个无符号字节(到整型寄存器),(从整型寄存器)存储一个字节
LH, LHU, SH	装载一个半字(到整型寄存器),装载一个无符号半字(到整型寄存器),(从整型寄存器)存储一个半字
LW, LWU, SW	装载一个字(到整型寄存器),装载一个无符号字(到整型寄存器),(从整型寄存器)存储一个字
LD, SD	装载一个双字(到整型寄存器),(从整型寄存器)存储一个双字
L.S, L.D, S.S, S.D	装载一个单精度浮点数,装载一个双精度浮点数,存储一个单精度浮点数,存储一个双精度浮点数
MFC0, MTC0	在通用寄存器和专用寄存器之间复制数据
MOV.S, MOV.D	复制一个单精度或双精度浮点数到另一个浮点寄存器
MFC1, MTC1	在浮点寄存器和整型寄存器之间复制32位数据
算术/逻辑指令	在GPR中对整型和逻辑数据进行操作;溢出会引发有符号算术陷阱
DADD, DADDI,	加,加立即数(所有的立即数为16位);有符号和无符号数
DADDU, DADDIU	
DSUB, DSUBU	减,有符号和无符号
DMUL, DMULU, DDIV,	乘和除,有符号和无符号;乘-加;所有操作数占用和生成64位值
DDIVU, MADD	
AND, ANDI	与,和立即数相与
OR, ORI, XOR, XORI	或,和立即数相或,异或,和立即数异或
LUI	装载高位立即数,即装载保存立即数的寄存器的32~47位,然后进行符号扩展
DSLL, DSRL, DSRA,	移位:立即数(DS_)和变量(DS_V)形式;包括逻辑左移、逻辑右移、算术右移
DSLLV, DSRLV, DSRAV	
SLT, SLTI, SLTU, SLTIU	有符号小于置位、有符号小于立即数置位;无符号小于置位、无符号小于立即数置位;
控制指令	条件转移和跳转;基于PC或通过寄存器
BEQZ, BNEZ	根据GPR内容等于/不等于0进行转移;16位位移量从PC+4开始
BEQ, BNE	根据GPR内容等于/不等于进行转移;16位位移量从PC+4开始
BCIT, BCIF	测试浮点数状态寄存器中的比较位然后转移,16位位移量从PC+4开始
MOVN, MOVZ	如果第三个GPR为负、0,将GPR的内容复制到另一个GPR中
J, JR	根据从PC+4开始的26位位移量(J)或寄存器中的目标(JR)跳转
JAL, JALR	跳转并链接:将PC+4保存到R31,转移目标是相对的PC(JAL)或某个寄存器(JALR)
TRAP	根据一个向量式地址转移到操作系统
ERET	从一个异常中返回到用户代码;恢复用户模式
浮点运算指令	在双精度和单精度格式上进行浮点操作
ADD.D, ADD.S, ADD.PS	双精度浮点数加,单精度浮点数加,一对单精度浮点数相加
SUB.D, SUB.S, ADD.PS	双精度浮点数减,单精度浮点数减,一对单精度浮点数相减
MUL.D, MUL.S, MUL.PS	双精度浮点数乘,单精度浮点数乘,一对单精度浮点数相乘
MADD.D, MADD.S,	双精度浮点数乘加,单精度浮点数乘加,一对单精度浮点数乘加
MADD.PS	
DIV.D, DIV.S, DIV.PS	双精度浮点数除,单精度浮点数除,一对单精度浮点数相除
CVT._._	转换指令:CVT.x.y从x类型转换到y类型,x和y是L(64位整型)、W(32位整型)、D(DP)或S(SP)。两个操作数都是FPR
C._.D, C._.S	双精度和单精度比较:“_”=LT,GT,LE,GE,EQ,NE;将浮点数状态寄存器中的相应位置位

译者序

我第一次看到美国 John L. Hennessy 教授和 David A. Patterson 教授所著的《计算机系统结构——量化研究方法（第一版）》一书是在 1991 年，当时，我正在德国汉诺威大学计算机系统结构和操作系统研究所做访问学者。我立刻被这本书的崭新内容吸引住了，恨不得立即将它据为己有，一“读”为快。可惜，此书当时实验室里只有一本，教授要用它备课，博士生要用它当参考书，粥少僧多，不能独占。想咬咬牙，自己买一本吧，谁知在汉诺威的书店里竟寻觅不到它的踪迹。当时，Internet 还没有像今天这样普及，网上购书更是没有的概念，所以只能插空，陆陆续续地读，直到 1992 年春回国，我也未能把它读完。

1992 年夏，周松年教授回国访问来到西安。当时正值他从多伦多大学拿到博士学位不久，刚刚自己创立了 Platform 公司。在和他的交谈中，我提到本书，流露出渴求的意思。过了一段时间，忽然接到松年从加拿大寄来的包裹，打开一看，正是渴望已久的书，当时的激动和喜悦至今难忘。于是，这本书成了我炫耀的宝贝，当然，在我的教学和科研生涯中，它发挥了莫大的作用。

再往后，本书出了第二版、第三版，又有了中译版，国内的条件也越来越好，我的那本第一版已不再是奇货可居。但是，我对它的感情却始终没变。

本书堪称是计算机系统结构学科的“圣经”。有多少学子在它的引导下进入了计算机系统结构的殿堂，从一无所知，到略知一二，到较为精通。本书不但有精辟的论述，还有大量习题，帮助学生理解、掌握书中所讲授的知识。多年来，本书在本科生和研究生教学中发挥的作用是不可估量的。更为难能可贵的是，两位作者笔耕不辍，每一次再版，都有新内容，给人以新的启示和教诲。这恐怕也是本书长盛不衰的原因。

此次第四版问世，更是给人以耳目一新的惊喜。全书内容经过重新安排，大量章节经过了重写。作者全面更新了与单核处理器到多核处理器历史发展相关的内容，以其独特的量化研究方法，阐述了多种体现多处理器系统结构优势的并行技术，这种“与时俱进”的态度令人肃然起敬。中国正处在研究开发我们自己的高性能通用处理器、向研制千万亿次高效能计算机冲刺的关键时期，本书第四版及其中译本的问世对正在从事这方面研究的中国学者来说，无疑是一个福音。

我与本书译者相识多年。他在师从西安交通大学郑守淇先生攻读计算机系统结构博士学位期间，就深受本书的影响。此次第四版中译本的问世，反映了他对本书的偏爱，也凝聚了他的很多心血。感谢他为读者做了一件大好事。我相信，严肃的读者定能从阅读本书中得到新的启示，吸取新的营养，在自己的事业中作出新的贡献。

钱德沛

2007 年 7 月 1 日于北京

作者简介

John L. Hennessy: 斯坦福大学校长, 1977年开始在斯坦福大学电子工程系和计算机系任教。他是IEEE和ACM会士, 美国国家工程院院士及美国科学与艺术院院士。由于其在RISC技术领域的杰出贡献, 于2001年被授予Eckert-Mauchly奖; 他获得的其他奖项还包括2001年度Seymour Cray计算机工程奖以及2000年度同David Patterson共同获得的John von Neumann奖。同时他还获得了荣誉博士学位。

1981年, Hennessy带领几个研究生开始其在斯坦福大学的MIPS项目。在1984年完成该项目之后, 他从学校离开了1年时间去开发一个MIPS计算机系统, 在开发该系统的过程中研制出了世界上第一个商用的RISC微处理器。MIPS Technologies公司从1991年被SGI公司收购, 后来在1998年脱离出来成为一个独立的公司, 并将其战略重点转向嵌入式微处理器。截至2006年, 已经有超过5亿个MIPS微处理器被用于视频游戏、掌上电脑、激光打印机和网络交换机等设备中。

David A. Patterson: 1977年开始在加州大学伯克利分校任职, 并讲授计算机系统结构课程。因其在教育方面的杰出贡献, 他荣获了Upsilon Pi Epsilon的Abacus奖、加州大学杰出教育奖、ACM颁发的Karlstrom奖、Mulligan教育奖章以及IEEE颁发的本科生教育奖。Patterson曾由于其对RISC技术的杰出贡献而获得IEEE技术贡献奖, 另外, 由于他在RAID技术方面的成就, 他赢得了IEEE Johnson信息存储奖。之后, 他和John L. Hennessy一起获得了John von Neumann奖。Patterson是美国科学与艺术院院士以及IEEE和ACM会士, 同时, 他被选入美国国家工程院、美国国家科学院以及硅谷工程名人堂。Patterson服务于美国总统信息技术顾问委员会, 同时也是伯克利大学EECS系计算机科学分部的主任, 计算研究协会主席以及ACM主席。这使他获得了CRA的杰出服务奖。

在加州大学伯克利分校, Patterson领导了RISC的设计与研发工作, 这几乎是第一台VLSI精简指令系统计算机。该研究工作成为后来SPARC系统结构的基础, SPARC系统结构目前被Sun微系统公司、富士公司及其他公司使用。他曾是RAID项目的领导者之一, 该项目使得许多公司得以应用其在可靠存储系统方面的研究成果。同时, 他还曾参加过网络工作站(NOW)项目, 成功研究出了目前许多Internet公司在使用的集群技术。上述这些项目获得了ACM颁发的三个论文奖。Patterson当前的研究项目在RAD实验室进行, 主要开发可靠、自适应、用于分布式Internet服务的相关技术; 另外, 还有多处理器研究加速器(RAMP)项目, 目的是开发基于FPGA的低成本、高可靠的并行计算机和开源硬件与软件。

序 言

Fred Weber, MetaRAM 公司总裁兼 CEO

我非常荣幸能够为这本在计算机系统结构领域极为重要的著作的第四版作序。在第一版中,我在产业界的第一位老师 Grodon Bell 就预言,该书将成为在计算机系统结构 and 设计领域中处于中心位置的一本经典著作。他的观点非常正确。我清晰地记得在该书刚出版时自己那种激动的心情。如今重读这本在后续三个版本中不断丰富的著作,我再次感受到了兴奋的感觉。坦率地说,在计算机系统结构领域中,甚至在任何我所了解的领域中,没有一本著作能够以这样迅速有效的方式使读者对计算机系统结构的知识从陌生到精通。

这本书数据翔实,图表清晰,理论与实践相结合,实例和描述丰富。该书用到了非常多的缩写、技术用语、趋势描述、公式、示例图和表格。这些对于计算机系统结构方面的著作来说再恰当不过。系统结构设计者的角色不像科学家或发明家那样,需要去深入研究一个特殊的现象或者创造新的基础性的材料或技术;也不像系统实施者那样,需要精通一些工具来精确地构建系统。系统结构设计者要做的工作应该是这样的:他们首先需要深刻理解最新的技术和做法,以及从过去到现在所期望的设计风格,然后他们需要拥有一种设计理念来和谐地构造完整的系统,最后要拥有信心和精力来合理搭配这些知识与资源从而完成构建工作。要做到这些,一名系统结构设计者需要大量的信息,同时深入理解基本概念并掌握定量的研究方法以便作为基础。而这正是这本书所要讲述的。

随着计算机系统的演变——从大型机、小型机和微处理器并存的时代,到以微处理器为主的时代,再到当前微处理器引入大型机的复杂技术的时代——Hennessy 和 Patterson 不断适时地更新这本著作。这本书的第一版介绍了 IBM 360, DEC VAX 以及 Intel 80x86, 这些是那一代计算机的典型代表。后续版本主要讨论了 80x86 和 RISC 处理器的细节,这在当时是处于统治地位的。最近的这一版本,讨论了线程与多处理技术、虚拟化与存储器层次结构以及存储系统,这为读者提供了一个与当前发展方向相吻合的背景,同时也为后十年的设计奠定了基础。这其中着重讨论了 AMD Opteron 和 SUN Niagara, 这是新时代多处理和 SoC 系统结构中, x86 与 SPARC(RISC)系统结构的典型代表,同时,它们也是创新的科技思想在真实的商业产品中的实际运用。

本书的第 1 章,用很少的篇幅介绍了计算机设计技术的分类以及计算机系统结构的基本概念,概述了驱动产业界发展的技术趋势,并给出了一种定量的方法帮助计算机系统设计。后两章主要讨论了传统 CPU 的设计,并深入讲述了该核心领域的一些可能方法和限制因素。最后三章主要讨论与多处理技术、存储器层次结构和存储系统相关的系统问题。这些领域的知识对于计算机系统结构设计者来说是非常重要的,同时当前片上系统设计时代,这些知识对每一个 CPU 设计者来说是必需的。最后,在附录中详细阐述了一些具体实例来帮助加深理解。

在设计工作中,同时以宏观和微观两种角度来分析问题并在这两种角度之间灵活转换的能力非常重要。在阅读本书时,读者会频繁地发现这两种角度。一个伟大的系统结构设计的成果,无论是计算机系统设计,还是建筑设计,抑或是教科书的设计,都要考虑客户的需求和期望并给出一个设计成果,使得客户能够满意地说:“哇,我不知道这样也是可行的。”本书在这一点上做得非常成功,我希望它可以给读者带来快乐和价值。

前言

关于此书的目的

从这本书的第一版开始一直到现在的第四版,我们都一直致力于对未来科技发展所基于的最基本原则进行阐述。我们对于计算机系统结构的热情从来都没有消退过,这里我们想引用第一版当中的一句话来说明:“这并不是什么徒有其表的纸模型,让人提不起兴趣。恰恰相反,这是一门研究人员相当感兴趣的学科,它需要在市场和成本性能之间找到平衡点。在人们寻找的过程中,既有可圈可点的失败案例,也有令人称赞的成功案例。”

在写这本书的第一版时,我们的初衷是希望改变人们对计算机系统结构原有的认知方式。现在,我们觉得这一初衷依然有存在的需要,而且相当重要。这一领域的变化日新月异,对它的研究必须基于真实计算机的真实案例和测量数据,简单地罗列一些永远都无法实现的定义和设计是毫无意义的。我们热情地欢迎我们的老朋友,同样也欢迎那些新加入的新朋友。不论怎样,我们都承诺对真实的系统采用相同的量化研究和分析方法。

和之前的版本一样,我们竭尽所能使这本书能够满足不论是专业工程师、系统架构师,还是那些参加高级计算机系统结构和设计课程的人的需求。与之前版本类似的还有一点,那就是本书希望能够通过强调成本、性能、功耗之间的关系以及优秀的工程设计来把看似复杂的计算机系统结构解释清楚。我们相信这一领域正在日趋成熟,而且会像其他经典的理工学科一样,将建立起强大的量化研究基础。

关于第四版

《计算机系统结构——量化研究方法》的第四版可能是其第一版以来最具特殊意义的一版。就在我们刚刚开始编写这一版不久,Intel 宣布它将与 IBM 和 Sun 一起研究单芯片上多处理器或多核所能实现的高性能设计。以下是这本书的第一组数字,在连续十六年保持每十八个月性能双倍提升的纪录之后,单核处理器性能改良的频率降到了每年幅度不大的提高。在计算机系统结构发展历程上的这一转折点意味着有史以来,第一次没有人能够给出速度更快的串行处理器。如果希望程序能够运行得更快,比如,对新增加的功能进行调整,那么就必须对程序进行并行处理。

因此,除了和前三个版本一样,继续关注指令级并行(ILP)所带来的高性能表现之外,这一版还对线程级并行(TLP)和数据级并行(DLP)给予了同样的关注。在之前的版本中,也有一些关于大型多处理器服务器上的TLP和DLP的内容,但是这一版会在单芯片的多核处理器中提到TLP和DLP。这一历史性的转变让我们相应更改了各个章节的顺序:在上一版本中第6章是关于多处理器的内容,在这一版本中将这一内容提前到第4章中来讲述。

不断发展的技术也促使我们将一些原本在稍后章节中的内容提前到第1章中。按照科学家们的预言,随着计算机产业向具有65 nm或更小特性的半导体处理器的方向发展,将导致更高的硬件和软件出错率。于是,我们决定将有关依赖性的一些基本概念从第三版中的第7章前移到第1章。除此之外,由于电源成为决定在一个芯片上能够放置多少内容的决定性因素,我们在第1章中增加了相关部分的内容。当然,所有章节的内容和举例都做了更新,下面将一一给出解释。

在这一版中,除了因技术领域的不断革新而带来内容的大量更新之外,我们还采用了新的方法来编写书中的习题。要编写出有趣、准确而且无歧义的习题是一件难度大、耗费时间的工作,而且这些习题必须要能够测试到一个章节中方方面面的内容。令人无奈的是,网络的使用使得这些习题的可用期减少了一半,最多只有几个月。学生不用自己费劲地找出答案,只需要在网络上搜索一下就能找到一本刚刚出版的教材上的习题的答案。于是,我们所付出的巨大努力很快便付之东流。对教师而言,他们也无法利用这些习题来检测学生对知识的掌握程度。

为了能够较好地解决这一问题,在这一版本中我们尝试了两个新的做法。第一,我们在学术界和业界聘请了各个主题的专家来编写这些习题。也就是说,每个领域的一些顶尖人物在帮助我们通过生动活泼的方式来探讨每个章节中的核心概念,并检测读者对于内容的理解程度。第二,每组习题都围绕一组范例分析展开。我们希望每个范例分析中的量化举例都能够在很长一段时期内保持其生命力。这些案例具有相当的弹性和足够的信息量,只要教师愿意,他们就能够方便地写出自己的练习题。当然,关键是每年我们都会为每个范例分析出版新的习题集。这些新的习题会在一些参数上做重大更改,这样老习题的答案就不再有效了。

另外一个重大的变化是我们沿袭了 *Computer Organization and Design* (COD)一书第三版的做法,将文字内容进行缩减,只包含任何一个读者都可能会希望看到的内容。对于那些某些读者认为是可有可无或者是参考内容的附录,我们把它放到了随书附带的光盘中。这样做,出于以下一些原因:

1. 学生抱怨这本书越来越重。第一版中主要章节的内容共有 594 页,另外还有 160 页的附录;到第二版,已经变成了 760 页的主要章节加上 223 页的附录;在第三版,这两个数字已经分别变成了 883 页和 209 页,除此之外,还有 245 页的在线附录^①。按照这样的速度,第四版将会有超过 1500 页的印刷内容和在线内容!
2. 基于同样的原因,教师们担心在这一门课程中会因内容过多而无法全部涵盖。
3. 和 *Computer Organization and Design* 一样,本书将部分内容放到了光盘中。它能使学生无论是否能够访问 Elsevier 的网站,都可以迅速地找到任何相关信息。而且,这一版中的附录即使在以后的版本出版以后,也能够继续为读者所用。
4. 这种灵活性让我们能够将关于流水线、指令系统、存储层次结构这些部分的回顾内容从各个章节里面抽取出来放到附录 A, B, C 中。对于教师和读者而言,这样做的好处在于他们可以快速地浏览这些回顾内容,然后将更多的时间和精力放到第 2 章、第 3 章和第 5 章的深层话题上。另外,我们还可以把一些重要但不是核心的课程内容放到光盘的附录中。这样一来,资料的量没有任何变化,但是印刷的书却精简了。在这一版中,我们一共有六个章节,所有的章节都不超过 80 页。在上一版中,我们一共有八个章节,最长的章节就有 127 页。
5. 这次的书加上光盘与以前的版本相比,制作成本更低。这样出版商也能够更多地降低售价。在目前的价格范围,已经没有必要为欧洲的读者单独制作一个海外学生版了。

还有一个与前一版本不同的重大变化是我们把第三版中的一些嵌入式系统资料移到了它自己的附录 D 中。我们觉得这些嵌入式系统材料并不总是和其余材料的量化评价方法相符合,而且还使得很多原本就已经很长的章节更加冗长。我们相信将所有嵌入式系统资料放到一个单独的附录中应该还会给教学带来一定的便利。

这一版本沿袭了使用真实世界中的实例来说明概念的做法。“综合”部分是全新的内容。事实上,有些内容是在我们这本书送交出版商之后才宣布的。这一版本中的“综合”部分包括了 Intel

^① 注意,本页的页数是指英文版的页数,不是指中文版的页数。——编者注

Pentium 4 和 AMD Opteron 的流水线组织和存储系统结构, 以及 Sun T1 (Niagara) 8 位处理器、32 位线程微处理器、最新的 NetApp Filer、Internet Archive 集群和 IBM Blue Gene/L 大规模并行处理器系统 (MPP)。

主题的选择和组织

和以前一样, 我们对主题的选择采取了保守的做法。毕竟, 在这个领域让人感兴趣的话题多如牛毛, 在这样一本以基本原理为核心的书中, 保守的做法是唯一合理的。我们没有对读者可能遇到的所有系统结构做详尽的调查说明。相反, 我们把重点放在那些任何一台新机器中都可能用到的核心概念上。我们的标准始终如一, 即选择那些已经被验证过且被成功实施过的技术, 这样才能对它们进行量化分析。

我们一直试图把精力放在那些无法从其他资源得到的内容上。因此只要有可能, 我们就不断地加强更高级的内容。事实上, 这本书里对好几个系统的描述都是首次公布 (对于想要了解关于计算机系统结构更为基础的一些概念的读者, 建议阅读 *Computer Organization and Design, The Hardware/Software Interface* 一书的第三版)。

内容概述

在这一版中, 第 1 章的内容更为丰富。其中包括了静态电源、动态电源、集成电路成本、可靠性以及可用性的计算公式。和此前的版本相比, 我们更深入地讨论了几何平均值和几何标准偏差在获取平均值变化方面的应用。我们希望这些概念可以在整本书中都得到体现和使用。除了经典的计算机设计和性能测量方面的量化原则之外, 对通用标准部分还进行了更新, 并使用了新的 SPEC2006 套件。

我们认为指令集系统结构 (ISAs) 在今天所扮演的角色已经没有 1990 年时那么重要了, 所以我们将这一部分内容移到了附录 B 中。其中使用的仍然是 MIPS64 系统结构。对于 ISAs 的支持者, 附录 J 涵盖了有关 RISC 系统结构、80x86、DEC VAX 和 IBM 360/370 方面的内容。

第 2 章和第 3 章讲述了指令级并行在高性能处理器中的运用, 其中包括超标量执行、转移^①预测、推测、动态调度以及相关的编译器技术。前文已经提到, 附录 A 对流水线进行了回顾, 需要时可作为参考。第 3 章对 ILP 的局限性进行了分析。这一版本中新增了对多线程的量化评估。第 3 章还包括了对以下不同处理器的逐项对比: AMD Athlon, Intel Pentium 4, Intel Itanium 2 和 IBM Power 5。这些处理器对 ILP 和 TLP 都有各自不同的应用。在上一版中有大量的篇幅对 Itanium 进行了讨论, 在这一版中, 这部分内容被移到了附录 G。原因是我们认为 Itanium 现在看来并没有当时所预计的那样有潜力。

由于业界现在已经从只对 ILP 进行研究转向了同时还研究线程级并行和数据级并行, 我们把与多处理器系统的相关内容提前到了第 4 章, 重点讲述共享存储式系统结构。这一章就是以这样一个系统结构的性能表现来引入主题的。这一章接着讨论了对称和分布存储层次结构, 研究了组织原则和性能。再下来是关于同步和存储一致性模型指令级并行的内容。其中用 Sun T1 (Niagara), 一个商业产品的初始设计来举例说明。这一设计回归到了单一指令发射、6 段流水微系统结构。它将 8 个这样的设计放到一个芯片上, 每个支持 4 个线程。于是软件会在这个低电源的单芯片上有 32 个线程。

上文已经提到, 附录 C 中对 Cache 原理进行了回顾, 需要时可作为参考。通过这一调整, 第 5 章可以一开始就讨论 11 种对 Cache 进行优化的方法。这一章新加入了有关虚拟机的内容。虚拟机在计算机保护、软件及硬件管理方面有着很多的优势。实例部分采用的是 AMD Opteron, 它最近刚刚扩展到了 64 位地址, 在 Cache 系统结构和虚拟内存模式方面很有代表性。

^① Branch 一词在全书中翻译为“转移”, 也可译为“分支”, 但译者认为前者更为贴切。——译者注

第6章进一步增加了对可靠性和可用性的讨论,并以 RAID 为例描述了 RAID 的 6 个模式,以及在真实系统上极少会发现的失效数据。接着介绍了排队论和 I/O 性能基准。与上一版本不同,我们没有一步一步地介绍构建一个假想集群的过程,取而代之的是对真实集群(Internet Archive)中成本、性能以及可靠性的评估。“综合”部分以 NetApp FAS60000 过滤器为例,它使用的是 AMD Opteron 微处理器。

下面是附录 A 到 L。上文已经提过,附录 A 和 C 是关于流水线和缓存概念的基本介绍。对于流水线还比较生疏的读者,应该在阅读第 2 章和第 3 章之前先阅读附录 A。对缓存不太了解的读者应该在阅读附录 C 之后再阅读第 5 章。

附录 B 讲述了 ISA 的基本原理,其中包括 MIPS64。附录 J 描述了 Alpha, MIPS, PowerPC 和 SPARC 的 64 位版本,以及它们各自的多媒体扩展。除此之外,这一部分还包括了一些经典的系统结构(80x86, VAX, IBM 360/370)以及比较主流的嵌入式指令系统(ARM, Thumb, SuperH, MIPS16、Mitsubishi M32R)。附录 G 也和这部分内容相关,主要讲述了 VLIW ISA 的系统结构。

附录 D 是由 Thomas M. Conte 来负责更新的,主要将嵌入的资料全部集中到了一起。

附录 E 是关于网络的。Timothy M. Pinkston 和 José Duato 对其做了大量的修改。附录 F 是由 Krste Asanovic 来修改的,其中包括了对向量处理器的描述。我们认为这两个附录是我们所了解的关于它们各自主题的最好的学习资料之一。

附录 H 描述了并行处理应用和一致性协议在大型共享存储的多处理过程中的应用。附录 I 是由 David Goldberg 完成的,介绍了计算机算术运算。

附录 K 将第三版中各个章节中的“历史回顾与参考”部分集中到了一起。这样做是为了更好地展示每个章节中的各种创新想法,让读者更好地了解这些发明背后的历史。我们希望呈现给读者一个有关计算机设计的演变进程。同时,它还给主修计算机系统结构的学生提供了很好的参考资料。如果有时间的话,我们建议读者可以读一些在这些章节当中提到的这一领域的经典文章。直接从创新者那里了解他们的想法是一件获益匪浅的乐事。“历史回顾”是以前版本中最受欢迎的章节。

附录 L(也可以通过 textbooks.elsevier.com/0123704901 访问)中是有关本书中范例分析习题的答案。

关于阅读顺序

阅读这本书没有一个最佳的顺序。但是所有的读者都应该从第 1 章开始阅读。如果不想通读这本书,则这里建议读者以这样一些顺序来阅读:

- ILP: 附录 A, 第 2 章和第 3 章, 附录 F 和 G
- 存储系统结构: 附录 C 和第 5 章、第 6 章
- 线程级和数据级并行: 第 4 章, 附录 H 和 E
- ISA: 附录 B 和 J

附录 D 可以随时阅读,但是在读完 ISA 和 Cache 后再阅读可能效果最好。在任何时候,只要读者对计算机算法感兴趣,都可以去参考附录 I。

关于章节结构

在每个章节里面,我们所选择的材料都是按照既定的结构进行组织的。每一章以这一章当中的主要概念介绍开始,接着是“相关问题”部分,它围绕着这一章中的内容是如何与其他章节中的内容相互关联而展开。第三部分称为“综合”,它通过真实的机器来把所有这些概念组织到一起。

第四部分是“谬误和易犯的错误”，其目的是希望学生能从别人的错误中汲取经验。我们给出的都是一些常见的误解和极难避免的系统结构陷阱。“谬误和易犯的错误”部分是这本书中最受欢迎的内容。每一章的内容均以“结论”结束。

范例分析及习题

每一章的最后都有范例分析和相关习题。这些范例和习题由业界和学术界的专家编撰而成，目的是对每一章的核心概念进行探讨，并通过越来越具挑战性的习题来考察理解的程度。教师们应该会发现这些范例分析都非常详尽，而且给予了他们编写自己的习题的很大空间。

每个习题中用尖括号括起的部分（<章节>）表示这部分内容对于完成习题非常重要。我们希望通过这种方式能让读者避免在没有阅读相关内容的情况下去试图做习题。而且这样读者也能够方便地找到需要回顾的内容。请注意所有习题的答案都在附录L中。我们根据每个习题可能需要的解答时间给习题划分了不同的等级：

- [10] 不到5分钟可以完成阅读和理解
- [15] 得出完整的答案需要5~15分钟
- [20] 得出完整的答案需要15~20分钟
- [25] 写出完整的答案需要1个小时
- [30] 小型编程项目：需要不到一天时间进行编程
- [40] 大型编程项目：需要两个星期的时间
- [讨论] 需要与其他人就主题进行讨论

那些在 textbooks.elsevier.com/0123704901 注册的教师，可以得到另外一套范例分析习题。这套习题会在每年的夏季做更新，这样到秋季新学期开始时，教师们就可以下载一套全新的配套习题和答案了。

补充材料

随书附带的光盘中包含了以下一些内容：

- 参考附录：一些由主题专家们撰写的猜想，其内容涵盖了许多高级话题
- 历史回顾资料：主要探讨了每一章中核心概念的发展历程
- 正文和光盘内容的搜索引擎

其他的附加内容可以在 textbook.elsevier.com/012374901 处访问到。而在教师专用站点（需要在 textbook.elsevier.com 注册），还包括以下内容：

- 另外一套完整的配套习题和答案（每年更新一次）
- 演示文档格式的教学演示稿
- 书中用到的图，JPEG 和 PPT 格式

在本书的配套网站（所有读者均可访问）上包括以下内容：

- 书中范例分析及习题的答案
- 相关资料的链接
- 勘误表

我们会定期对新的资料和网络上相关资源的链接进行更新。

给出修改建议

最后,基于与成本-性能类似的考虑,当读者在读这本书时,别忘了还可以赚钱!如果读者看看下面的“致谢”这一部分,就会了解我们在更正错误方面所做出的努力。因为这本书经过了多次出版,我们可以有很多机会来改正其中的错误。如果读者发现了任何遗留的错误,请通过电子邮件和出版商联系(邮件地址为 ca4bugs@mkp.com)。第一位报告错误且给出更正的读者,如果更正在我们以后的出版当中被采纳,这位读者将获得 1 美元的奖励。请在报告错误之前浏览我们的主页(textbooks.elsevier.com/0123704901)上的勘误表,查看该错误是否已经被报告过。我们会处理这些错误并在大概一年之内寄出奖金的支票,请读者耐心等待。

我们同时也欢迎对于这本书的任何建议和意见,请将你的意见发送到邮箱 ca4comments@mkp.com。

结束语

这里我们想再次强调的是这本书完全是合作的结果,我们每个人负责编写每章以及附录的一半内容。我们无法想象如果没有另外的人来负责另一半的工作,这本书将会需要多久才能完成。我们彼此感谢对方在自己感到任务的完成遥遥无期时所给予的激励,当自己遇到晦涩的概念时提供解释这些概念的好方法,在周末花时间帮对方做内容的审核,当在其他工作的繁重压力下自己感到无法再提起笔时对方给予的安慰(随着这本书的一版再版,其他的工作急剧增多。特别是我们其中一个人是斯坦福大学的校长,而另外一位是美国计算机学会的主席)。当然,我们也共同承担读者因书中的内容而产生的任何不满和抱怨。

John L. Hennessy 和 David A. Patterson

致 谢

虽然这本书仅仅只有四个版本,但是实际上我们有了九个不同版本:第一版有三个版本(alpha版、beta版和最终版),第二版有两个版本,再加上第三版和第四版(beta版和最终版)。在此期间,我们得到了数百位审阅者和用户的帮助,是他们使得这本书更加完善。为此,我们决定列出所有对这本书不同版本做出贡献的人。

第四版贡献者

跟以前的版本一样,这是包括许多志愿者在内的集体的努力。没有他们的帮助,这个版本就不可能如此优秀。

审阅者

麻省理工学院的 Krste Asanovic; 密歇根大学的 Mark Brehob; 弗吉尼亚大学的 Sudhanva Gurumurthi; 威斯康星大学麦迪逊分校的 Mark D. Hill; 伊利诺伊州立大学香槟分校的 Wen-mei Hwu; 西北大学的 David Kaeli; 得克萨斯大学奥斯汀分校的 Ramadass Nagarajan; 得克萨斯大学奥斯汀分校的 Karthikeyan Sankaralingam; 克莱姆森大学的 Mark Smotherman; 威斯康星大学麦迪逊分校的 Gurindar Sohi; 印第安纳州圣母大学的 Shyamkumar Thoziyoor; 弗吉尼亚大学的 Dan Upton; 新泽西理工学院的 Sotirios G. Ziavras。

核心团队

麻省理工学院的 Krste Asanovic; 西班牙瓦伦西亚理工大学的 José Duato; Intel 西班牙加泰罗尼亚理工大学的 Antonio González; 威斯康星大学麦迪逊分校的 Mark D. Hill; 瑞尔森大学的 Lev G. Kirischian; 南加州大学的 Timothy M. Pinkston。

附录

麻省理工学院的 Krste Asanovic (附录 F); 北卡特莱纳大学的 Thomas M. Conte (附录 D); 西班牙瓦伦西亚理工大学的 José Duato (附录 E); 施乐帕洛阿尔托研究中心的 David Goldberg (附录 I); 南加州大学的 Timothy M. Pinkston (附录 E)。

习题和答案贡献者

威斯康星大学麦迪逊分校的 Andrea C. Arpaci-Dusseau 和 Remzi H. Arpaci-Dusseau (第 6 章); R&E Colwell & Assoc 公司的 Robert P. Colwell (第 2 章); 加州理工大学圣路易斯-奥比斯波分校的 Diana Franklin (第 1 章); 伊利诺伊州立大学香槟分校的 Wen-mei W. Hwu (第 3 章); 惠普实验室的 Norman P. Jouppi (第 5 章); 伊利诺伊州立大学香槟分校的 John W. Sias (第 3 章); 威斯康星大学麦迪逊分校的 David A. Wood (第 4 章)。

附加材料

John Mashey (第 1 章中的几何平均值和标准偏差); 加州大学伯克利分校的 Chenming Hu (第 1 章中的 wafer 开销和 yield 参数); AMD 的 Bill Brantley 和 Dan Mudgett (第 5 章中的 Opteron 存储器结构层次性能); Mendel Rosenblum, Stanford 和 Vmare (第 5 章的虚拟机); Aravind Menon,

EPFL Switzerland (第5章的Xen度量); Bruce Baumgart 和 Brewster Kahle, Internet 存储档案 (第6章的IA簇); David Ford, Steve Kleiman 和 Steve Miller, 网络应用 (第6章中的FX6000信息); Rutgers 公司的 Alexander Thomasian (第6章的排队论)。

最后,我们要特别感谢克莱森姆大学的 Mark Smotherman, 他详细阅读了本书的修订稿, 找出书中非常多的错误, 使得最终版本的错误大大减少。

当然, 如果没有出版商该书也不会出版。所以感谢 Morgan Kaufmann/Elsevier 全体职员的努力和支持。对于第四版, 我们要特别感谢 Kimberlee Honjo, 他协调了调查、核心组、草稿评审和附录整个过程, 以及 Nate McFadden, 他负责这本书的整个过程和习题的审查。我们还要感谢我们的主编 Densie Penrose, 他的领导使我们能够把我们的工作继续下去。

我们还要感谢学校里的职员 Margaret Rowland 和 Cecilia Pracher, 他们为我们处理了无数的传真和邮件, 并在我们撰写本书的过程中辅助我们在斯坦福(大学)和伯克利(加州大学伯克利分校)的事务。

最后我们要感谢我们的妻子, 感谢她们对我们无数个长夜和清晨的阅读、思考还有打字输入的忍耐和宽容。

先前版本的贡献者

审阅者

普度大学的 George Adams; 伊利诺伊州立大学香槟分校的 Sarita Adve; 布里格姆青年大学的 Jim Archibald; 麻省理工学院的 Krste Asanovic; 华盛顿大学的 Jean-Loup Baer; 西北大学的 Paul Barr; 得克萨斯大学圣安东尼奥分校的 Rajendra V. Boppana; 得克萨斯大学奥斯汀分校的 Doug Burger; SGI 的 John Burger, Michael Butler, Thomas Casavant, Rohit Chandra; 密歇根州立大学的 Peter Chen; 纽约州立大学石溪分校、卡内基梅隆大学、斯坦福大学、克莱姆森大学和威斯康星大学的课程; Vitesse Semiconductor 公司的 Tim Coe; Intel 公司的 Bob Colwell, David Cummings, Bill Dally, David Douglas; 东南密苏里州立大学的 Anthony Duben; 华盛顿大学的 Susan Eggers, Joel Emer, Dartmouth 的 Barry Fagin, David Filo; 惠普实验室的 Josh Fisher, DIKU 的 Rob Fowler; 华盛顿大学的 Mark Franklin, Kourosh Gharachorloo; 哈佛大学的 Nikolas Gloy; 施乐帕洛阿尔托研究中心的 David Goldberg; 威斯康星大学麦迪逊分校的 James Goodman; 哈佛穆德学院的 David Harris; John Heinlein; 斯坦福大学的 Mark Heinrich; 加州大学圣克鲁斯分校的 Daniel Helman; 威斯康星大学麦迪逊分校的 Mark Hill; IBM 公司的 Martin Hopkins; 惠普实验室的 Jerry Huck; 宾夕法尼亚州立大学的 Mary Jane Irwin, Truman Joe, Norm Jouppi; 西北大学的 David Kaeli; 内布拉斯加大学的 Roger Kiechhafer; 普度大学的 Allan Knies, Don Knuth; 斯坦福大学的 Jeff Kuskin; 微软研究院的 James R. Larus; 多伦多大学的 Lori Liebrock, Hank Levy; 普林斯顿大学的 Kai Li; 阿拉斯加费尔班克斯塔纳纳谷大学的 Lori Liebrock; 威斯康星大学麦迪逊分校的 Mikko Lipasti; 北卡罗来纳大学教堂山分校的 Gyula A. Mago; 伍斯特理工学院的 William Michalson, James Mooney; 密歇根大学的 Trevor Mudge; 卡内基梅隆大学的 David Nagel, Todd Narter, Victor Nelson; 加州大学伯克利分校的 Vojin Oklobdzijia; 斯坦福大学的 Kunle Olukotun; 宾夕法尼亚州立大学的 Bob Owens; Sun 公司的 Greg Papadapoulous, Joseph Pfeiffer; 康奈尔大学的 Keshav Pingali; 滑铁卢大学的 Bruno Preiss, Steven Przybylski, Jim Quinlan, Andras Radics; 乔治亚理工学院的 Kishore Ramachandran; 得克萨斯大学奥斯汀分校的 Joseph Rameh; 康奈尔大学的 Anthony Reeves; 密歇根州立大学的 Richard Reid; 密歇根大学的 Steve Reinhardt; 加州大学洛杉矶分校的 David Rennels; 麻萨诸塞州立大学阿默斯特分校的 Arnold L. Rosenberg; 普度大学的 Kaunshik Roy; Unysis 公司的 Emilio

Salqueiro, Petere Schnorf, Margo Seltzer; 南卫理公会大学的 Behrooz Shirazi; 卡内基梅隆大学的 Daniel Siewiorek; 普林斯顿大学的 J. P. Singh, Ashok Singhal; 威斯康星大学麦迪逊分校的 Jim Smith; 哈佛大学的 Mike Smith; 克莱姆森大学的 Mark Smotherman; 威斯康星大学麦迪逊分校的 Guri Sohi; 华盛顿大学的 Arun Somani; 克莱姆森大学的 Gene Tagliarin; 俄勒冈大学的 Evan Tick; 北卡莱罗纳州大学教堂山分校的 Akhilesh Tyagi; Universidad Politécnic de Cataluña, Barcelona 的 Mateo Valero; 加州大学圣克鲁斯分校的 Anujan Varma; 康奈尔大学的 Thorsten von Eichen; 得克萨斯 A&M 大学的 Hank Walker; 施乐帕洛阿尔托研究中心的 Roy Want; Sun 公司的 David Weaver; 特拉维夫大学的 Shlomo Weiss, David Wells; 卡内基梅隆大学的 Mike Westall, Maurice Wilkes; 普度大学的 Thomas Wills; Malcolm Wing; 纽约州立大学石溪分校的 Larry Wittie; 乔治亚理工学院的 Witte Zegura。

附录

麻省理工学院的 Krste Asanovic 修订了向量附录; 浮点附录由施乐帕洛阿尔托研究中心的 David Goldberg 编写。

习题

普度大学的 George Adams; 威斯康星大学麦迪逊分校的 Todd M. Bezenek (他要借此工作纪念他的祖母 Ethel Eshom); Susan Eggers; Anoop Gupta; David Hayes; Mark Hill; 加州大学圣克鲁斯分校的 Ethan L. Miller; 原康柏公司西部研究实验室的 Parthasarathy Ranganathan; 威斯康星大学麦迪逊分校的 Brandon Schwartz, Michael Scott, Dan Siewiorek, Mike Simth, Mark Smotherman, Evan Tick, Thomas Willis。

特别感谢

美国国防部高级研究计划署的 Dnane Adams, Tom Adams; 伊利诺伊州立大学香槟分校的 Sarita Adve, Anant Agarwal; 罗切斯特大学的 Dave Albonesi, Mitch Alsup, Howard Alt, Dave Anderson, Peter Ashenden, David Bailey; 美国国防部高级研究计划署的 Bill Bandy; 原康柏公司西部研究实验室的 L. Barroso, Andy Bechtolsheim, C. Gordon Bell, Fred Berkowitz, IBM 公司的 John Best, Dileep Bhandarkar; BDTI 公司的 Jeff Bier, Mark Birman, David Black, David Bogg, Jim Brady, Forrest Brewer; 加州大学伯克利分校的 Aaron Brown; 原康柏公司西部研究实验室的 E. Bugnion; 罗切斯特大学的 Alper Buyuktosunoglu, Mark Callaghan, Jason F. Cantin, Paul Carrick, Chen-Chung Chang; 罗切斯特大学的 Lei chen, Peter Chen, Nhan Chu; 普林斯顿大学的 Dong Clark, Bob Cmelik, John Crawford, Zarka Cvetanovic; 得克萨斯大学奥斯汀分校的 Mike Dahlin, Merrick Darley; 原 DEC 公司西部研究实验室的员工 John DeRosa, Lloyd Dickman, J. Ding; 华盛顿大学的 Susan Eggers; 罗切斯特大学的 Wael El-Essawy; Mills 的 Patty Enriquez, Milos Ercegovac, Robert Garner; 原康柏公司西部研究实验室的 K. Gharachorloo, Garth Gibson, Ronald Greengerg, Ben Hao; 原康柏公司的 John Henning; 威斯康星大学麦迪逊分校的 Mark Hill, Danny Hillis, David Hodges; Google 公司的 Urs Hoelzle, David Hough, Ed Hudson; 伊利诺伊州立大学香槟分校的 Chris Hughes, Mark Johnson, Lewis Jordan, Norm Jouppi, Willian Kahan, Randy Katz, Ed Kelly, Richard Kessler, Les Kohn; 原康柏公司的 John Kowaleski, Dan Lambright; Sun 微系统公司的 Gary Lauterbach, Corinna Lee; Ruby Lee, Don Lewine, Chao-Huang Lin; 美国国防部高级研究计划署的 Paul Losleben, Yung-Hsiang Lu; 美国国防部高级研究计划署的 Bob Lucas, Ken Lutz; Intel 伯克利研究实验室的 Alan Mainwaring; Rutgers 公司的 Rich Martin, John Mashey, Luke McDowell; Trimedia 公司的

Sebastian Mirolo; Ravi Murthy; Biswadeep Nag; Sun 微系统的 Lisa Noordergraaf; 美国国防部高级研究计划署的 Bob Parker; Internet 研究中心的 Vern Paxson, Lawrence Prince, Steven Przybylski; 美国国防部高级研究计划署的 Mark Puller, Chris Rowen, Margaret Rowland; 罗切斯特大学的 Greg Semeraro, Bill Shannon, Behrooz Shirazi, Robert Shomler, Jim Slager; 克莱姆森大学的 Mark Smotherman; 华盛顿大学的 SMT 研究小组; 美国国防部高级研究计划署的 Steve Squires, Ajay Sreekanth, Darren Staples, Charles Stapper, Jorge Stolfi, Peter Stoll; 第一次使用本书的斯坦福和伯克利的学生 Bob Supnik, Steve Swanson, Paul Taysom, Shreekant Thakkar; 新泽西理工学院的 Alexander Thomasian; 美国国防部高级研究计划署的 John Toole, Trimedia 公司的 Kees A. Vissers, Willa Walker, David Weaver; EMC 的 Ric Wheeler, Maurice Wilkes, Richard Zimmerman。

John L. Hennessy 和 David A. Patterson

目 录

第 1 章 计算机设计基本原理	1
1.1 简介	1
1.2 计算机的分类	3
桌面计算机	3
服务器	4
嵌入式计算机	5
1.3 计算机系统结构的定义	5
指令集系统结构	6
系统结构的其他方面: 设计满足目标和功能要求的组成和硬件	8
1.4 实现技术的发展趋势	9
性能的发展趋势: 带宽优于时延	10
晶体管性能与连线的规模	12
1.5 集成电路功耗的发展趋势	12
1.6 成本的发展趋势	13
时间、产量、产品化的影响	14
集成电路的成本	15
成本与价格	17
1.7 可靠性	18
1.8 测量、报告和总结计算机的性能	20
基准测试程序	20
性能评价报告	23
性能评测结果的总结	23
1.9 计算机设计的量化原则	26
采用并行性	26
局部性原理	27
关注经常性事件	27
Amdahl 定律	27
处理器性能公式	29
1.10 综合: 性能和性价比	32
桌面计算机和机架式系统的性能和性价比	32
事务处理服务器的性能和性价比	33
1.11 谬误和易犯的错误	35
1.12 结论	38
1.13 历史回顾和参考文献	39

1.14	范例分析及习题	39
	范例分析 1: 芯片制作成本	39
	范例分析 2: 计算机系统功耗	40
	范例分析 3: Web 服务器中可靠性 (及故障) 的开销	42
	范例分析 4: 性能	43
第 2 章	指令级并行及其开发	45
2.1	指令级并行: 概念与挑战	45
	什么是指令级并行	46
	数据相关和冒险	47
2.2	支持指令级并行的基本编译技术	51
	基本流水线调度和循环展开	51
	循环展开和调度的小结	54
2.3	采用预测技术减小转移开销	55
	静态转移预测	55
	动态转移预测和转移预测缓存	56
	Tournament 预测器: 整体局部自适应预测器	59
2.4	采用动态调度克服数据冒险	61
	动态调度: 概念	62
	用 Tomasulo 方法进行动态调度	63
2.5	动态调度: 举例和算法	67
	Tomasulo 算法: 细节	69
	Tomasulo 算法: 一个基于循环的例子	69
2.6	基于硬件的推测	72
2.7	采用多发射和静态调度技术开发指令级并行	79
	基本的 VLIW 方法	80
2.8	采用动态调度、多发射和推测方法开发指令级并行	82
2.9	指令传送和推测的高级技术	84
	提高取指令带宽	84
	推测: 实现问题和扩展	88
2.10	综合: Intel Pentium 4	91
	Pentium 4 性能分析	92
2.11	谬误和易犯的错误	97
2.12	结论	98
2.13	历史回顾和参考文献	99
2.14	范例分析及习题	99
	范例分析 1: 探讨微系统结构技术的影响	99
	范例分析 2: 对转移预测器建模	104
第 3 章	指令级并行性的限制	106
3.1	介绍	106
3.2	指令级并行性限制的研究	106

硬件模型	107
窗口大小和最大发射数的限制	108
实际转移和跳转预测的影响	110
有限寄存器的影响	112
非完美别名分析的影响	113
3.3 实际处理器中的指令级并行性限制	114
克服研究模型的限制	117
3.4 相关问题: 硬件推测和软件推测	118
3.5 多线程: 使用指令级并行支持线程级并行的开发	119
同时多线程: 将线程级并行转换为指令级并行	120
3.6 综合: 高级多发射处理器的性能和效率	124
是什么限制了多发射处理器	126
3.7 谬误和易犯的错误	127
3.8 结论	128
3.9 历史回顾和参考文献	128
3.10 范例分析及习题	128
范例分析 1: 相关和指令级并行	128
第 4 章 多处理器和线程级并行	134
4.1 简介	134
并行系统结构的分类	135
通信和存储器系统结构模型	138
并行处理遇到的挑战	138
4.2 对称式共享存储器系统结构	140
什么是多处理器的 Cache 一致性	140
实施一致性的基本方案	142
监听协议	143
基本实现技术	143
协议范例	144
对称式共享存储器多处理器和监听协议的局限性	148
实现监听 Cache 一致性	149
4.3 对称式共享存储器多处理器的性能	149
商业负载	150
商业负载的性能测试	151
多道程序和操作系统负载	155
多道程序和操作系统负载的性能	156
4.4 分布式共享存储器和基于目录的一致性	158
基于目录的 Cache 一致性协议: 基础知识	159
目录协议范例	161
4.5 同步: 基本要素	163
基本硬件原语	164

107	用一致性实现锁	165
108	4.6 存储器连贯性模型：介绍	167
110	程序员的视角	168
112	非严格连贯性模型：基本概念	168
113	关于连贯性模型的最后小结	169
114	4.7 相关问题	169
117	编译器优化和连贯性模型	169
118	在严格连贯性模型中用推测来实现时延隐藏	169
119	包含性及其实现	170
120	4.8 综合：Sun T1 多处理器	171
124	T1 的性能	171
126	运行 SPEC 基准测试程序的多核处理器的性能	175
127	4.9 谬误和易犯的错误	177
128	4.10 结论	181
128	4.11 历史回顾和参考文献	181
128	4.12 范例分析及习题	182
128	第 5 章 存储器层次结构设计	198
134	5.1 简介	198
134	5.2 11 种先进的 Cache 性能优化方法	202
135	第一种优化：小而简单的 Cache 减少命中时间	203
138	第二种优化：路预测减少命中时间	204
138	第三种优化：踪迹 Cache 减少命中时间	204
140	第四种优化：流水线 Cache 访问	205
140	第五种优化：利用非阻塞 Cache 增加 Cache 带宽	205
142	第六种优化：利用多组 Cache 增加 Cache 带宽	206
143	第七种优化：关键字优先和提前重启动以减小缺失代价	207
143	第八种优化：合并写缓冲区以降低缺失代价	207
144	第九种优化：编译器优化以降低缺失率	209
148	第十种优化：指令和数据硬件预取以降低缺失代价 / 缺失率	211
149	第十一种优化：编译控制预取降低缺失代价 / 缺失率	212
149	Cache 优化技术小结	214
150	5.3 存储器技术及性能优化	214
151	SRAM 技术	215
155	DRAM 技术	215
156	在 DRAM 芯片内部改善存储器性能	217
158	5.4 保护：虚拟存储器和虚拟机	218
159	通过虚拟存储器来提供保护	219
161	虚拟机的保护	220
163	虚拟机监视器的必备条件	221
164	虚拟机（缺乏）的指令集系统结构支持	221

虚拟机在虚拟存储器和 I/O 上的冲突	222
虚拟机监视器的实例: Xen 虚拟机	222
5.5 相关问题: 存储器层次设计	225
保护和指令集系统结构	225
预测执行和存储系统	225
I/O 和 Cache 数据的一致性	226
5.6 综合: AMD Opteron 存储器层次结构	226
Opteron 存储层次结构的性能	230
5.7 谬误和易犯的错误	233
5.8 结论	238
5.9 历史回顾和参考文献	239
5.10 范例分析及习题	239
范例分析 1: 通过简单的硬件实现 Cache 性能优化	239
范例分析 2: 通过先进技术优化 Cache 性能	241
范例分析 3: 存储器技术及优化	242
范例分析 4: 虚拟机	243
范例分析 5: 综合: 高度并行化存储系统	245
第 6 章 存储系统	248
6.1 简介	248
6.2 磁盘存储的高级话题	248
磁盘功耗	250
磁盘阵列的高级话题	251
6.3 实际故障的定义和实例	254
伯克利 Tertiary Disk 系统	255
Tandem	256
其他方面研究: 操作员在可靠性中扮演的角色	257
6.4 I/O 性能可靠性评测	258
吞吐率与响应时间	259
事务处理基准测试程序	260
SPEC 系统级文件服务器、邮件及 Web 的基准测试程序	261
基准测试程序可靠性的实例	262
6.5 排队论简介	263
随机变量的泊松分布	266
6.6 相关问题	272
点到点连接和用交换机代替总线	272
块服务器与文件管理器的对比	272
异步 I/O 和操作系统	273
6.7 I/O 系统设计与评价——互联网存储档案集群	273
互联网存储档案集群	274
互联网存储档案集群的成本、性能和可靠性评价	274

222	计算 TB-80 集群的 MTTF	276
222	6.8 综合: NetApp FAS6000 文件管理器	277
225	6.9 谬误和易犯的错误	278
225	6.10 结论	282
225	6.11 历史回顾和参考文献	282
226	6.12 范例分析及习题	282
226	范例分析 1: 解析磁盘	282
230	范例分析 2: 解析磁盘阵列	284
233	范例分析 3: RAID 重构	287
238	范例分析 4: RAID 性能的预测	289
239	范例分析 5: I/O 子系统设计	290
239	范例分析 6: 失效位	291
239	范例分析 7: 排序	294
241	附录 A 流水线: 基础和中级概念	297
242	附录 B 指令系统原理与实例	352
243	附录 C 存储器层次结构回顾	385
245	参考文献	427
248	索引	453
248		
250		
251		
254		
255		
256		
257		
258		
259		
260		
261		
262		
263		
266		
272		
272		
272		
273		
273		
274		
274		

第1章 计算机设计基本原理

某些内容已完全不同。

——Monty Python's Flying Circus

1.1 简介

自从第一台通用电子计算机研制成功以来,计算机技术在近60年的时间里取得了令人瞩目的发展。如今,我们花不到500美元购买的个人计算机就比1985年花100万美元购买的计算机具有更高的性能、更大的存储器和磁盘空间。这一高速发展既得益于计算机实现技术的进步,同时也离不开计算机设计方法的创新。

尽管计算机实现技术一直稳步发展,但新的系统结构的发展却总是相对滞后。在电子计算机诞生后的最初25年,由于实现技术和系统结构的发展,计算机系统的性能以每年25%的速度提高。20世纪70年代末,集成电路技术的进步和微处理器的出现为计算机性能的进一步提升注入了新的推动力——计算机性能以大约每年35%的速度提高。

35%的年增长率以及微处理器大批量生产的成本优势,使得以微处理器为基础的相关产业迅速发展。此外,计算机市场出现了两个重大变化:其一,人们实际已经极少使用汇编语言编程,这就降低了对目标代码兼容性的要求;其二,通用、跨平台操作系统(如UNIX和Linux)的出现,将促使新的计算机系统结构的成本和风险进一步降低。上述变化使得新的计算机系统结构比以前更容易实用化和产业化。

正是由于这些变化,在20世纪80年代初相继推出了一系列包含更简单指令的新系统结构,称为RISC(精简指令系统计算机)。基于RISC的计算机主要关注两种关键的实现技术,即指令级并行(从采用流水线到使用多指令发射)和Cache(高速缓存)的使用(从初期简单的组织形式到后来更加复杂的结构和优化方式)。

RISC计算机成为当时系统结构发展的主流。DEC的VAX系统结构由于没能及时跟上这个发展步伐,因而被RISC系统结构所取代。而Intel则接受了这个挑战,在处理器内部将x86(或IA-32)指令翻译成类RISC指令,并采用了许多在RISC设计中首先被提出来的创新方法。自20世纪90年代后期,计算机中芯片的集成度的增加使得翻译更加复杂的x86指令所导致的硬件开销几乎可以忽略不计了。

图1.1说明了近16年来系统结构和组成技术的发展所带来的性能以每年超过50%的持续增长,该增长率在计算机产业中是空前的。

计算机性能在20世纪的飞速发展带来了双重效果。一方面,它给用户提供了强大的功能。在许多应用中,目前的高性能微处理器的性能已经超过了几年前超级计算机的性能。

另一方面,这一飞速发展使得基于微处理器的计算机在整个计算机领域占据了统治地位,工作站和PC已经成为计算机产品的主流。基于逻辑电路或门阵列的小型机已经被基于微处理器的服务器所取代。由一定数量的微处理器构成的多处理器系统已经取代了大型机,甚至高端的超级计算机也可以由多个高性能微处理器构成。

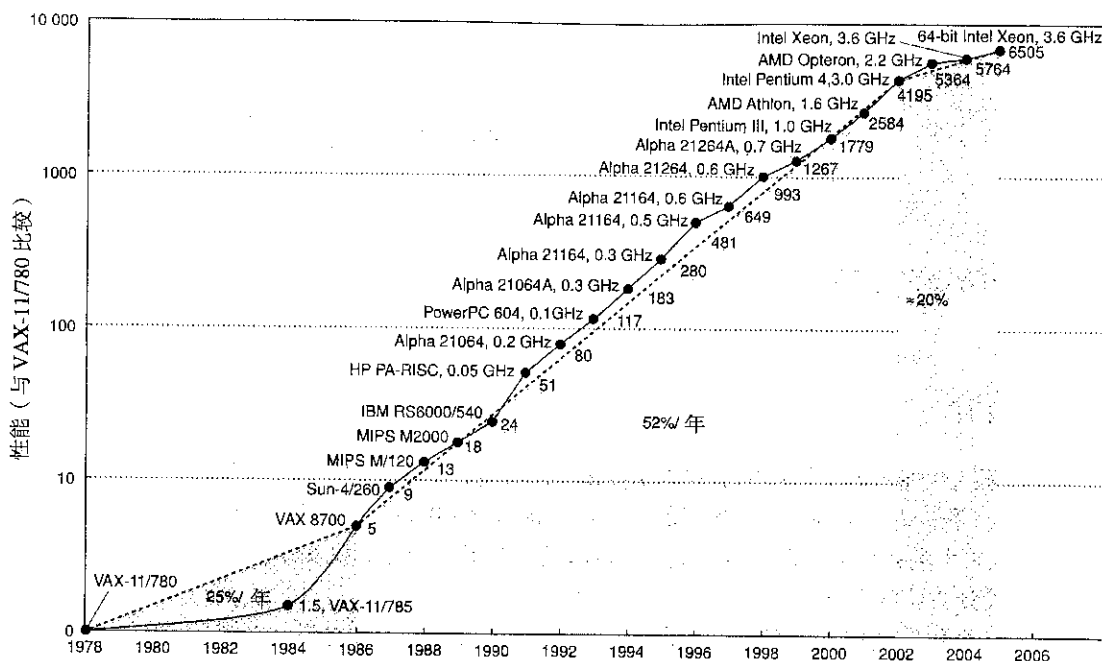


图 1.1 20 世纪 80 年代中期以来处理器性能的增长。此图是根据 SPECint 基准程序测得的数据而绘制的计算机性能曲线图，以 VAX11/780 为基准。在 20 世纪 80 年代中期以前，处理器的性能改进主要依赖于实现技术的提高，平均以大约每年 25% 的速率增长。自 20 世纪 80 年代中期开始，由于系统结构和组成技术的发展，该增长达到了 52%。到 2002 年为止，性能提高了近 7 倍。针对浮点计算的性能增长得更快。自 2002 年开始，由于在电源、可用的指令级并行和存储器长时延等几方面的限制，使近几年单一处理器性能提升的速率减缓到大约每年 20%。由于 SPEC 基准测试程序的改进，对于新机器来说，其性能是通过参照两个不同版本的 SPEC（如 SPEC92，SPEC95 和 SPEC2000）的换算系数来评估的

上述技术创新带来了计算机设计的又一次飞跃，此时的计算机设计既强调系统结构的创新，又充分利用实现技术的成果。在此推动下，至 2002 年，高性能的微处理器比单独依靠实现技术（包括改进电路设计）所能达到的性能要高出 7 倍之多。

然而，图 1.1 同样也表明计算机这一段长达 16 年的飞速发展阶段已告结束。自 2002 开始，处理器性能的增长率已经降到了每年 20% 的水平。造成这种现象的原因包括：风冷芯片的最大功耗；所剩无几的可以有效发掘的指令级并行；难以降低的存储器时延。事实上，在 2004 年，Intel 已取消了其高性能单一处理器的研究计划，转而加入到 IBM 和 Sun 的阵营中来，并宣称其将通过同一芯片上的多处理器而不是更快的单一处理器来进一步提高计算机系统的性能。这标志着一个历史性转折的到来，处理器性能的改进不能再只依赖于指令级并行（ILP），即本书前三版的主要关注内容，而应更加关注线程级并行（TLP）或是数据级并行（DLP），即本书的主要内容。因为即使没有程序员的干预，编译器和硬件也能够充分利用指令级并行机制，相比起来，线程级并行和数据级并行属于显式并行，因为它们均要求程序员编写并行代码以改善系统性能。

本书主要从系统结构的角度和相关编译器的改进方面来分析计算机系统性能在 20 世纪快速增长的原因所在。此外，我们还格外关注现在和未来上述两方面所面临的挑战及潜在的方法和技术。本书中的核心内容是计算机设计量化研究方法的发展及其使用的分析方法——对程序的经验观察。

实验和模拟。本书所体现的正是这样一种设计风格和研究方法。本书的目的不仅在于讲述这种设计方式,更在于希望能够激励读者为计算机技术的发展做出贡献。同时,我们也相信量化研究方法正如对过去的隐式并行计算机有很大帮助一样,它对未来的显式并行计算机的设计也会做出积极的贡献。

1.2 计算机的分类

20世纪60年代,在计算机领域中占统治地位的是大型机。这需要花费上百万美元和专门放置计算机的房间,此外还需要多个操作者监视运行。这种大型机的典型应用包括商业数据处理和大规模科学计算。20世纪70年代见证了小型机的诞生,起初它主要针对实验室中的科学应用,但是其应用范围很快就扩展到了分时系统——多个用户通过各自独立的终端交互地共享一台计算机。同一时期也出现了面向科学计算的高性能计算机——超级计算机。虽然数量不多,但是它在计算机的历史上却占据了重要的地位,这是因为它在设计上首次采用了多项创新技术,这些技术后来被逐渐应用到了其他较廉价的计算机上。20世纪80年代则是桌面计算机崛起的时代,这种基于微处理器的计算机,分为个人计算机和 workstation 两种。这种个人化的桌面计算机取代了传统的分时系统,并导致了服务器的产生。服务器是可以提供可靠的长时间文件存储和访问、大容量的存储器以及更强的计算能力等大规模服务的计算机。20世纪90年代诞生了 Internet 和万维网、第一台实用的掌上计算设备(个人数字助理或 PDA)以及从视频游戏机到机顶盒等各种高性能数字消费电子设备。自2000年开始,手机开始大范围流行起来,其功能改进和销售额的增长率都超过了 PC。目前的许多新的应用都是依靠嵌入式计算机来完成的。

上述演变使我们在新千年的开始对计算技术、计算应用和计算机市场等方面的观点发生了巨大改变。在个人计算机诞生20余年后的今天,我们亲历了计算机在外观以及使用方法上的巨大变化,这些变化推动了三个细分产品市场的形成,它们中的每一个都拥有各不相同的应用、需求和技术。图1.2总结了这些计算环境的主流类别及其重要特征。

特征	桌面计算机	服务器	嵌入式计算机
系统价格	500~5000 美元	5000~5 000 000 美元	10~100 000 美元 (包括高端网络路由器)
微处理器组件价格	50~500 美元 (每个处理器)	200~10 000 美元 (每个处理器)	0.01~100 美元 (每个处理器)
系统设计的关键问题	性价比、 图形性能	吞吐量、可用性、 可量测性	价格、能量消耗、 特定应用的性能

图 1.2 三大主流计算机技术的分类和各自的特征。此图记录了服务器和嵌入式系统的价格范围。对于服务器来说,这个范围的增长来自于对于超大规模多处理器系统的需求。这种系统主要用于高端事务的处理过程和 Web 服务器的应用。如果算上 8 位和 16 位微处理器,2005 年嵌入式处理器的总销售数目接近 30 亿。同年销售了约 2 亿台桌面计算机和 1 千万台服务器

桌面计算机

首个也是最大的一个市场即为桌面计算机市场。桌面计算机的范围涵盖了从低于 500 美元的低端计算机到超过 5000 美元、拥有超高配置的工作站。在这个价格和性能区间,计算机市场的总体趋势是提高其性价比。性能(主要通过计算性能和图形性能来进行衡量)和价格的综合因素是消费者最关心的,因此也就成为设计者关注的焦点。因此,桌面计算机往往是最新、最高性能的微处理器和低成本微处理器最先应用的领域(见 1.6 节有关影响计算机成本因素的讨论)。

虽然基于 Web 和交互式的应用对系统性能提出了更高的要求，但桌面计算机在应用和基准测试方面有其独特之处。

服务器

在桌面计算机流行的同时，服务器在提供更大规模及更可靠文件与计算服务方面的重要性也日趋显现。万维网的出现加速了这种趋势，这是因为对 Web 服务器和基于 Web 的业务的需求在快速增长。这些服务器取代了传统的大型机进而成为企业进行大规模信息处理的中枢。

对服务器而言，如下特性是至关重要的。首先，可靠性是关键（将在 1.7 节讨论可靠性）。比如运行 Google、处理 Cisco 业务、或者在 eBay 上进行拍卖业务的服务器，必须确保每周 7 天、每天 24 小时连续运转。如果这样的服务器系统出现故障，那么其后果比一台桌面计算机的故障所带来的损失更具灾难性。图 1.3 给出了 2000 年以来由于服务器停机所造成的收入损失。根据最新统计，Amazon 网站在 2005 年秋季的销售额为 29.8 亿美元，在该季度约 2200 小时的情况下，平均每小时收入为 135 万美元。在圣诞节购物的高峰期，如果服务器出现停机，其带来的潜在损失会更高。

因此可以看出，系统如果不可用，其估测成本是很高的，而且图 1.3 仅仅给出了收入上的损失，这还没有包括由于顾客的不满而造成的损失。

应用	每小时损失 (千美元)	停机时间段内年平均损失 (百万美元)		
		1% (87.6 小时/年)	0.5% (43.8 小时/年)	0.1% (8.8 小时/年)
经纪人业务	6450	565	283	56.5
信用卡核查	2600	228	114	22.8
货运服务	150	13	6.6	1.3
家庭购物频道	113	9.9	4.9	1.0
目录销售中心	90	7.9	3.9	0.8
航空预定中心	89	7.9	3.9	0.8
手机服务激活	41	3.6	1.8	0.4
网络在线费用	25	2.2	1.1	0.2
ATM 服务费	14	1.2	0.6	0.1

图 1.3 通过分析停机时间段的开销（直接收入损失）得到的系统不可用的损失数据，假设有三种不同级别的有效性，且停机时间段是均匀分布的。数据来自 Kenbel[2000]，由 Contingency Planning Research 公司整理和分析

服务器系统的第二个关键特性是可扩展性。由于服务需求或功能需求的增长，服务器也应随之扩展。因此对于服务器来讲，能够在计算能力、存储器容量、存储系统以及 I/O 带宽等方面进行升级是至关重要的。

最后，服务器的主要设计目标就是为了达到高效的吞吐量，也就是说，服务器的整体性能——以每分钟处理的事务数或每秒所提供的页面数来衡量——是关键所在。对单个请求的响应非常重要，但用单位时间处理的请求数目来表示整体效率和成本效率对大多数服务器来说更为关键。我们将在 1.8 节再次讨论不同类型计算环境下的性能评估问题。

一类和服务器紧密相关的是超级计算机。通常它们价格昂贵，一般要花费几千万美元，且更注重浮点运算的性能。附录 H 中讨论的集群，已在很大程度上取代了这一类计算机。随着集群的流行，传统的超级计算机及其制造商的数量均在减少。

嵌入式计算机

嵌入式计算机是计算机市场中增长最快的领域。这种智能设备在日常生活中随处可见,其范围涵盖从日常使用的电器(大部分的微波炉、洗衣机、打印机、网络交换机以及所有的小汽车都含有简单的嵌入式微处理器)到手持数据设备(如手机和智能卡),以及视频游戏机和数字机顶盒。

嵌入式计算机的处理能力和价格覆盖的范围很广:从低端售价低于10美分的8位和16位处理器,到低于5美元、每秒可以执行1亿条指令的32位微处理器,还有高端的价值100美元、能够为最新的视频游戏机或高端网络交换机提供每秒10亿条指令计算能力的嵌入式处理器。尽管嵌入式计算机处理能力的范围非常广,但价格仍然是设计计算机时需要重点考虑的一个关键因素。设计的主要目标是以最低的价格满足实际的性能需求,而不是追求用更高的价格来实现更高的性能。

通常,嵌入式应用的性能需求具有**实时性**,即每个程序段有一个确定的最大执行时间。例如,在数字机顶盒中,由于处理器必须在很短的时间内接收和处理下一帧,因此,每一个视频帧的处理时间是有限的。在一些应用中有着更加复杂的需求,即最大时间范围内一个特定任务的平均时间和例程的数目是受限的。为此提出了一种**软实时**的方法,这种方法能够在必要时忽略一个事件的时间限制,同时又不放弃其他更多的约束。实时性能往往与更高要求的应用密切相关。

很多嵌入式应用还有其他两个关键特性:最小化存储器需求和最小化功耗需求。在很多嵌入式应用中,存储器是系统成本的一部分,因此,对存储器大小的优化是很重要的。有时应用程序能够装入处理器芯片上的内存,而有时应用程序则需要装入一个片外存储器。无论采用哪种方式,由于数据大小是由应用程序决定的,因此,存储器大小的问题就转化成了代码量大小的问题。

较大容量的存储器同样意味着更高的功耗,而优化功耗本身在嵌入式应用中同样也是关键问题。尽管由于使用电池是格外关注功耗的一个重要原因,但使用更便宜的封装方式(塑料封装或陶瓷封装)的需求和省却制冷风扇的需求都要限制整体的功耗。我们将在本章后续部分(1.5节)详细介绍这个问题。

本书的大部分内容都适用于嵌入式处理器(不管是一般的微处理器还是需要和其他专用硬件组合在一起的微处理器内核)的设计、使用和性能分析。

实际上本书的第三版已经包含了许多嵌入式计算的例子,用以阐述每一章节的观点。不过多数读者会发现这些例子并不令人满意,究其原因是对桌面计算机和服务器的量化设计和分析时的数据没有很好地涵盖嵌入式计算机(比如,可参看1.8节中的EEMBC挑战)。基于此,本书中不再保留嵌入式计算机的有关内容,而把嵌入式的相关材料整理成单独的一篇附录。我们相信这一新的(附录D)将会进一步增强本书内容的完整性,并有助于使读者理解不同的需求对嵌入式计算所产生的影响。

1.3 计算机系统结构的定义

计算机设计者所面临的任务异常复杂:对一台新计算机而言,首先要判断哪些特征是最重要的,然后,在不超过成本、电源和可用性限制的范围内力求性能最优。这里面涵盖的问题可能是多方面的,包括指令系统的设计、功能结构、逻辑设计与实现。其中实现技术可能涉及集成电路的设计、封装、电源的设计以及冷却措施。优化设计需要熟练掌握很多领域的技术,从编译器、操作系统到逻辑设计和封装技术。

过去,计算机系统结构通常是指指令系统设计,计算机设计的其他方面则称为实现。这种说法通常暗示着计算机实现技术不引人关注或不具有挑战性。

我们确信这种说法是不正确的。系统结构设计者的工作不仅仅是指令系统设计,在其他方面遇到的技术困难比在指令系统设计中遇到的更有挑战性。首先我们简要回顾一下指令集系统结构,随后再介绍这些计算机系统结构所面临的挑战性问题。

指令集系统结构

在本书中,指令集系统结构 (ISA, instruction set architecture) 指的是程序员可见的实际指令系统。ISA 的作用相当于硬件和软件之间的一个分界面。我们通过 MIPS 和 80x86 两个例子从 ISA 的以下七个方面简要回顾一下指令集系统结构。

1. **ISA 分类:** 现在几乎所有的 ISA 都被归类为通用的寄存器系统结构,其操作数或者是寄存器,或者是存储器地址。80x86 拥有 16 个通用寄存器和 16 个支持浮点运算的寄存器,而 MIPS 则拥有 32 个通用寄存器和 32 个浮点寄存器 (如图 1.4 所示)。这一类 ISA 包含两种不同的形式,一种是诸如 80x86 这样的 register-memory 式 ISA,可以通过多种指令访问存储器;另一种是 MIPS 这样的 load-store 式 ISA,只能通过装载和存储指令访问存储器。近期所有的 ISA 都是 load-store 式 ISA。

名称	编号	用途	调用时保存?
\$zero	0	常量值为 0	N.A.
\$at	1	临时汇编	否
\$v0-\$v1	2-3	函数返回值和表达式赋值	否
\$a0-\$a3	4-7	函数调用参数	否
\$t0-\$t7	8-15	临时变量	否
\$s0-\$s7	16-23	可保存的临时变量	是
\$t8-\$t9	24-25	临时变量	否
\$k0-\$k1	26-27	操作系统预留	否
\$gp	28	全局指针	是
\$sp	29	堆栈指针	是
\$fp	30	帧指针	是
\$ra	31	返回地址	是

图 1.4 MIPS 寄存器及使用规范。除了 32 个通用寄存器 (R0~R31) 之外, MIPS 还拥有 32 个浮点寄存器 (F0~F31), 这些寄存器既可以用于 32 位的单精度数运算,也可以用于 64 位的双精度数运算

2. **存储器寻址:** 实际上,包括 80x86 和 MIPS 在内的所有桌面计算机和服务器都是使用字节形式来访问存储器中的操作数的。一部分系统结构,像 MIPS 等,要求对象必须是对齐的,例如要访问字节地址 A 处 s 字节大小的操作数时,如果 A 可以被 s 整除,则说明地址 A 是对齐的 (见附录 B 中的图 B.5)。80x86 不要求地址对齐,但如果操作数是对齐的,那么一般来说访问会更加快捷。
3. **寻址方式:** 除了特殊的寄存器和值为常量的操作数之外,寻址方式需要明确说明操作数的地址。MIPS 的寻址方式有寄存器寻址、立即数寻址和相对寻址 (即将一个常数偏移量和寄存器相加得到存储器地址)。80x86 支持上述寻址方式和三种变址寻址方式:无寄存器 (绝对寻址)、有两个寄存器 (基址寻址)、两个寄存器且其中一个的内容和操作数的字节数大小相乘 (变址寻址)。它还支持类似于上述但要减去偏移数的寻址方式:寄存器间接寻址、基址寻址和变址寻址。
4. **操作数的类型和大小:** 就像大多数 ISA 一样, MIPS 和 80x86 支持的操作数大小有 8 位 (ASCII 码), 16 位 (Unicode 码或者半字), 32 位 (整型数或字), 64 位 (双字或者长整型数) 以及

32 位（单精度）和 64 位（双精度）的 IEEE 754 浮点数。80x86 还支持 80 位的浮点数（双倍扩展精度）。

5. 操作指令：操作指令一般分为数据传输指令、算术逻辑运算指令、控制指令（后面讨论）以及浮点数操作指令。MIPS 是一种简单且易于实现流水的指令集系统结构，也代表了 2006 年流行的 RISC 系统结构。图 1.5 总结了 MIPS 的 ISA。80x86 则拥有更加丰富的指令系统（见附录 J）。

指令类型	指令含义
数据传输	在寄存器和存储器之间，或者在定点和 FP 或特殊寄存器之间传递数据；存储器地址仅为 16 位的偏移量加上通用寄存器（GPR）的内容
LB, LBU, SB	装载字节，无符号字节，存储字节（从/到定点寄存器）
LH, LHU, SH	装载半字，无符号半字，存储半字（从/到定点寄存器）
LW, LWU, SW	装载字，无符号字，存储字（从/到寄存器）
LD, SD	装载双字，存储双字（从/到定点寄存器）
L.S, L.D, S.S, S.D	装载 SP、DP 浮点数，存储 SP、DP 浮点数
MFC0, MTC0	从/到 GPR 复制内容到/从特殊寄存器
MOV.S, MOV.D	将 SP 或 DP 浮点寄存器的内容复制到另一个浮点寄存器
MFC1, MTC1	从/到定点寄存器复制 32 位到/从浮点寄存器
算术/逻辑	对 GPR 中的整型或逻辑型数据进行操作；带符号算术运算溢出时进入陷阱
DADD, DADDI, DADDU, DADDIU	加，加立即数（均为 16 位）；有符号，无符号
DSUB, DSUBU	减，有符号，无符号
DMUL, DMULU, DDIV, DDIVU, MADD	乘和除，有符号和无符号；乘-加；所有操作都是针对 64 位数的操作
AND, ANDI	与，与立即数相与
OR, ORI, XOR, XORI	或，与立即数相或，异或，与立即数异或
LUI	16 位立即数填入目标寄存器高 16 位，目标寄存器的低 16 位填 0
DSLL, DSRL, DSRA, DSLLV, DSRLV, DSRAV	双字逻辑左移，双字逻辑右移，双字算术右移，可变双字逻辑左移，可变双字逻辑右移，可变双字算术右移
SLT, SLTI, SLTU, SLTIU	若小于则设定，若小于立即数则设定；有符号，无符号
控制	条件转移和跳转；相对 PC 寄存器或通过寄存器
BEQZ, BNEZ	GPRs 等于/不等于 0 转移；16 位偏移量在 PC+4 地址中
BEQ, BNE	GPR 相等/不等转移；16 位偏移量在 PC+4 地址中
BC1T, BC1F	测试 FP 状态寄存器中的比较位并转移；16 位偏移量在 PC+4 地址中
MOVN, MOVZ	如果第三个 GPR 的内容是负数/0，则复制 GPR 的内容到另一个 GPR
J, JR	跳转：26 位偏移量在 PC+4 地址(J)或目标寄存器 JR 中
JAL, JALR	跳转和关联：将 PC+4 地址保存在 R31 中，目标相对于 PC(JAL)或寄存器(JALR)
TRAP	根据地址向量转入管态
ERET	从异常中返回到用户代码；恢复用户态
浮点	对 DP 和 SP 数据的浮点操作
ADD.D, ADD.S, ADD.PS	DP 和 SP 操作数相加，一对 SP 操作数相加
SUB.D, SUB.S, SUB.PS	DP 操作数减 SP 操作数，一对 SP 操作数相减
MUL.D, MUL.S, MUL.PS	DP 和 SP 操作数相乘，一对 SP 操作数相乘
MADD.D, MADD.S, MADD.PS	DP 和 SP 操作数相乘加，一对 SP 操作数相乘加
DIV.D, DIV.S, DIV.PS	DP 操作数除以 SP 浮点数，一对 SP 操作数相除
CVT	变换指令：CVT.x.y 从 x 类型变换到 y 类型，x 和 y 可以为 L（64 位整数）、W（32 位整数）、D（DR）或 S（SP）。两个操作数都为 FPRs
C.C.D, C.C.S	DP 和 SP 比较：“_”=LT, GT, LE, GE, EQ, NE；置位 FP 状态寄存器

图 1.5 MIPS64 的子指令系统。SP 代表单精度，DP 代表双精度。附录 B 中提供了更多的关于 MIPS64 的细节。对于数据来说，最高位是第 0 位，最低位是第 63 位

6. 控制流指令: 实际上, 包括 80x86 和 MIPS 在内的所有 ISA 都支持条件转移指令、无条件跳转指令、程序调用和返回指令。80x86 和 MIPS 都使用相对 PC 寄存器的寻址, 即将指令的地址码与 PC 寄存器内容相加后形成转移地址。但两者也略有不同: MIPS 中的条件转移指令 (BE, BNE 等) 检测寄存器中的内容, 而 80x86 的转移指令 (JE, JNE 等) 检测根据算术/逻辑指令操作结果而设置的条件码位; MIPS 的过程调用指令 (JAL) 将返回地址存放在寄存器中, 而 80x86 的过程调用指令 (CALLF) 将返回地址存放在存储器堆栈中。
7. ISA 的编码: 主要有两种编码选择——固定长度和可变长度。所有的 MIPS 指令都是 32 位长度的, 这样就简化了指令的译码。图 1.6 表明了 MIPS 指令的格式。80x86 的编码是可变长的, 长度范围从 1 位到 18 位。可变长度的指令比固定长度的指令占据的空间要少, 因此为 80x86 编译的程序通常比同样的为 MIPS 编译的程序要小。上述几个方面通常会对指令怎样进行二进制编码产生影响, 例如, 由于寄存器字段和寻址方式字段会在一条单一的指令中出现多次, 因此, 寄存器数量和寻址方式的数量会对指令的大小产生显著的影响。

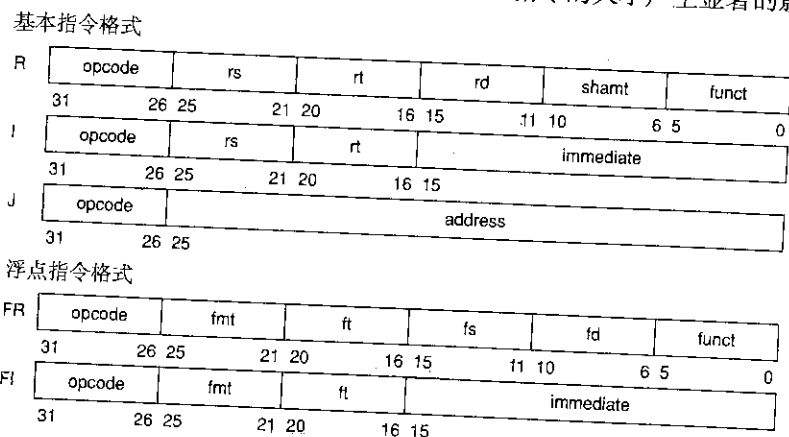


图 1.6 MIPS64 指令集结构格式。所有的指令都是 32 位长的。R 格式用于定点寄存器之间的操作, 例如 DADDU, DSUBU 等。I 格式用于数据传输、转移和立即数指令, 例如 LD, SD, BEQZ 和 DADDI 等。J 格式用于跳转指令、浮点运算中的浮点数格式以及浮点转移指令中的 FI 格式

当不同指令系统之间的差异不大且应用领域不同时, 除指令集系统结构涉及设计外, 计算机系统结构设计者所面临的其他挑战越发突出, 所以除了在本节对指令系统做简要介绍外, 更多有关指令系统的内容都放在附录 (附录 B 和附录 J) 中。

在本书中我们使用 MIPS64 的子集作为指令集系统结构的例子。

系统结构的其他方面: 设计满足目标和功能要求的组成和硬件

一台计算机的实现包括两部分内容: 组成和硬件。组成涵盖了计算机设计的更高层次, 例如存储系统、存储器互连以及内部的处理器或 CPU (中央处理单元——实现算术、逻辑、转移和数据传输指令) 的设计。例如, AMD Opteron 64 和 Intel Pentium 4 拥有相同的指令集系统结构, 但却有着完全不同的组成。两者都执行 x86 指令系统, 但却有完全不同的流水线和 Cache 结构。

硬件是一台计算机的具体实现技术, 包括具体的逻辑设计和封装技术。同一系列的计算机通常具有相同的指令集系统结构和几乎完全相同的组成, 但它们的具体硬件实现却不同。例如, Pentium 4 和移动 Pentium 4 几乎是完全相同的, 但两者提供了不同的时钟频率和存储系统, 结果使得移动 Pentium 4 更适用于低端计算机。

在本书中,系统结构一词包含了上述计算机设计的所有三个方面——指令集系统结构、组成和硬件。

计算机系统结构的设计者必须设计能满足包括价格、供电、性能和可用性指标的计算机。图 1.7 总结了在设计计算机时需要考虑的因素。一般而言,首要任务是要明确功能需求,其中包括适应市场需求的特定功能。通过确定如何使用机器,应用软件通常决定了对特定功能要求的选择。如果市场上已有大量为某一指令集系统结构设计的软件,系统结构设计者就应考虑在新的机器中与这一指令系统兼容。如果对某一特定应用软件的市场需求非常大,设计者也可能会在新的计算机中引入某些支持这些软件的功能特性,以增强新计算机在市场中的竞争力。其中许多需求和特性将在后续章节中深入讨论。

功能需求	应具备或支持的典型特性
应用领域	计算机的目标
通用桌面计算机	对一系列任务有较均衡的性能,包括图像、视频和音频的交互性能(第2章、第3章、第5章、附录B)
科学计算桌面计算机和服务器	具有较高的浮点运算和图像处理功能(附录I)
商用服务器	支持数据库和事务处理功能;有较强的可靠性和可用性;支持可扩展性(第4章、附录B、附录E)
嵌入式计算机	通常要求对图像、视频或是其他一些专门的应用有特殊的支持;要求功耗限制和控制(第2章、第3章、第5章、附录B)
软件兼容程度	决定计算机可以运行软件的数量
编程语言	设计者自由度大;需要新的编译器(第4章、附录B)
目标代码或二进制兼容	系统结构已经确定;自由度小;但无须对软件或端口程序追加投入
操作系统要求	为支持选定的操作系统所必需的特性(第5章、附录E)
地址空间的大小	非常重要的特性(第5章),可能限制应用程序
存储器管理	现代操作系统的需要;可以是页式或段式(第5章)
保护	不同的操作系统和应用都需要考虑:页式或段式;虚拟机(第5章)
标准	市场已经有某种需要的特定标准
浮点	格式和算法:IEEE 754 标准(附录I),针对图像或信号处理的特殊算法
I/O 接口	I/O 设备:串行 ATA, 串行 Attach SCSI, PCI Express(第6章、附录E)
操作系统	UNIX, Windows, Linux, CISCO IOS
网络	对不同网络的支持:以太网, Infiniband(附录E)
编程语言	语言(ANSI C, C++, Java, FORTRAN)影响指令系统(附录B)

图 1.7 系统结构设计者要考虑的重要性能需求一览表。左边第一列是需求的分类,右边一列给出了特定的例子,同时也包括了与该问题相关的章节和附录

系统结构设计者还应关注实现技术和计算机应用方面的重要发展趋势,因为这不仅影响到机器未来的成本,也影响到所设计的系统结构的生存周期。

1.4 实现技术的发展趋势

如果一个指令集系统结构的设计能够适应计算机技术的快速发展,那么它才能称得上是成功的。毕竟一个成功的指令可能会延续使用数十年,比如 IBM 大型机已经使用了超过 40 年的时间。系统结构的设计应考虑未来技术的发展,以延长计算机的生存周期。

从计算机的长远发展考虑,设计者必须了解计算机实现技术的快速变化。以下四种实现技术的飞速发展对现代计算机技术的影响最为深远:

- **集成电路技术:** 晶体管的密度以大约每年 35% 的速度增长,差不多每四年翻两番。芯片尺度过增长速度较慢且较难预测,大约在每年 10%~20% 之间。其综合效果是,每个芯片上的晶体管数目以每年大约 40%~55% 的速度增加。器件速度的增长较慢,这部分内容将在后面讨论。

- **半导体 DRAM (动态随机存取存储器)**: 其容量每年增长约 40%, 每两年翻一番。
- **磁盘技术**: 1990 年以前, 其密度每年增长约 30%, 每三年翻一番。在此之后, 增长速度提高 to 每年 60%, 特别是自 1996 年更是达到了每年 100%。从 2004 年开始又回落到每年 30%。尽管磁盘实现技术经历了起伏跌宕的发展过程, 但每比特位的成本仍然要比 DRAM 便宜 50~100 倍。我们将在第 6 章详细讨论该技术的发展趋势。
- **网络实现技术**: 网络的性能取决于交换和传输系统的性能。我们将在附录 E 讨论网络发展的趋势。

这些快速发展的技术, 可以将计算机设计的生存周期延长至 5 年或更长。即使在单个计算系统的生存周期中 (两年设计期, 两到三年投产期), 设计者也必须考虑关键技术 (如 DRAM) 的巨大变化所带来的影响。事实上, 计算机设计所采用的实现技术应具有一定的前瞻性, 因为产品批量生产时正是这些技术最有成本效益或性能最优时。通常, 成本降低的速度与器件密度提高的速度基本相当。

这些技术的变化尽管是连续的, 但它们的影响有时却是跳跃式的, 这种阶梯式变化推动了系统实现更高的性能。例如, 20 世纪 80 年代初期, 当 MOS 技术发展到一个硅片上可以集成 25 000~50 000 个晶体管时, 制造单片的 32 位微处理器就成为可能。到 20 世纪 80 年代末, 一级 Cache 在芯片上出现。通过消除处理器之间以及处理器和 Cache 之间的芯片交叉, 可以极大地提高性价比和性能/功耗比。但这种设计在技术未达到一定水平之前是难以实现的。这样的技术突破对诸多设计都有很大的影响。

性能的发展趋势: 带宽优于时延

就像 1.8 节中将介绍的那样, 带宽或吞吐量指的是在给定时间内所完成的工作总量, 例如, 在磁盘传输数据时每秒可以传输多少兆字节的数据。与此不同, 时延或响应时间则是指从事件开始到完成所需要的时间, 例如访问一次磁盘需要多少毫秒。图 1.8 给出了对各个微处理器、存储器、网络和磁盘技术发展的里程碑而言, 带宽和时延的相对改进。图 1.9 描述了上述示例和里程碑的细节。显而易见, 带宽的提高速度明显快于时延的提高速度。

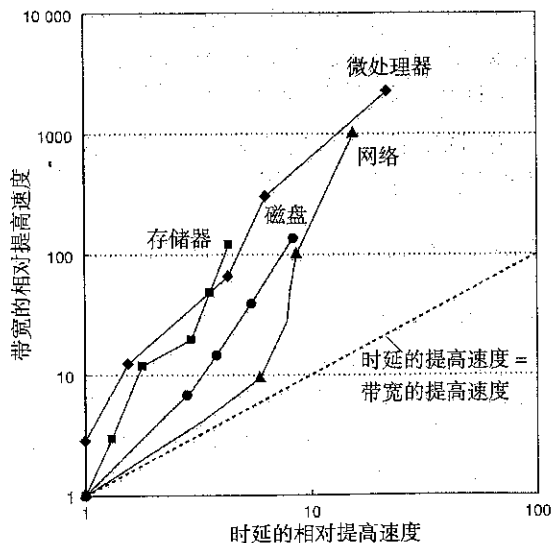


图 1.8 相对于第一个技术里程碑, 图 1.9 中的里程碑事件在带宽和时延上的改进, 尺度以对数表示。可以看出, 每当时延改善 10 倍, 带宽就会提高 100 至 1000 倍 (源自 Patterson [2004])

微处理器	16-bit address/bus, microcoded	32-bit address/bus, microcoded	5-stage pipeline, on-chip I & D caches, FPU	2-way superscalar, 64-bit bus	Out-of-order 3-way superscalar	Out-of-order superpipelined, on-chip 1.2 cache
产品	Intel 80286	Intel 80386	Intel 80486	Intel Pentium	Intel Pentium Pro	Intel Pentium 4
年	1982	1985	1989	1993	1997	2001
硅片面积 (平方毫米)	47	43	81	90	308	217
晶体管	134 000	275 000	1 200 000	3 100 000	5 500 000	42 000 000
管脚	68	132	168	273	387	423
时延 (时钟)	6	5	5	5	10	22
总线宽度 (比特)	16	32	32	64	64	64
时钟频率 (兆赫)	12.5	16	25	66	200	1500
带宽 (MIPS)	2	6	25	132	600	4500
时延 (纳秒)	320	313	200	76	50	15
存储器模块	DRAM	Page mode DRAM	Fast page mode DRAM	Fast page mode DRAM	Synchronous DRAM	Double data rate SDRAM
模块宽度 (比特)	16	16	32	64	64	64
年	1980	1983	1986	1993	1997	2000
兆比特/DRAM 芯片	0.06	0.25	1	16	64	256
硅片面积 (平方毫米)	35	45	70	130	170	204
管脚/DRAM 芯片	16	16	18	20	54	66
带宽 (兆比特/秒)	13	40	160	267	640	1600
时延 (纳秒)	225	170	125	75	62	52
局域网	Ethernet	Fast Ethernet	Gigabit Ethernet	10 Gigabit Ethernet		
IEEE 标准	802.3	803.3u	802.3ab	802.3ac		
年	1978	1995	1999	2003		
带宽 (兆比特/秒)	10	100	1000	100 00		
时延 (微秒)	3000	500	340	190		
硬盘	3600 rpm	5400 rpm	7200 rpm	10 000 rpm	15 000 rpm	
产品	CDC WrenI 94145-36	Seagate ST41600	Seagate ST15150	Seagate ST39102	Seagate ST373453	
年	1983	1990	1994	1998	2003	
容量 (GB)	0.03	1.4	4.3	9.1	73.4	
磁盘类型参数	5.25 inch	5.25 inch	3.5 inch	3.5 inch	3.5 inch	
介质直径	5.25 inch	5.25 inch	3.5 inch	3.0 inch	2.5 inch	
接口	ST-412	SCSI	SCSI	SCSI	SCSI	
带宽 (兆比特/秒)	0.6	4	9	24	86	
时延 (毫秒)	48.3	17.1	12.7	8.8	5.7	

图 1.9 在 20 到 25 年间, 微处理器、存储器、网络 and 磁盘性能发展的里程碑。微处理器的里程碑是六代 IA-32 处理器, 从 16 位总线、微程序实现的 80286 到 64 位总线、超标量系统结构、乱序执行、超流水线的 Pentium 4。存储器模块的里程碑是从 16 位普通的 DRAM 到 64 位双倍数据速率的同步 DRAM, 即 DDR SDRAM。以太网从 10 Mb/s 发展到 10 Gb/s。磁盘的里程碑是基于转速的, 从 3600 rpm 进步到 15 000 rpm。每个例子都给出了最优的带宽, 而时延是假定不发生冲突的情况下完成简单操作的时间 (源自 Patterson[2004])

性能是微处理器和网络的主要指标, 它们在性能方面有最大的提高: 带宽有了 1000 倍到 2000 倍, 时延改进了 20 倍到 40 倍; 而对于存储器和磁盘来说, 容量比性能更加重要, 所以其容量的提高最快, 它们在带宽方面提高了 120 到 140 倍, 比起 4 到 8 倍的时延改进来说还是大得多。显然, 带宽的改进要优于时延, 而且这一趋势仍将继续。

一种简单的经验法则就是带宽的提高速度至少相当于时延改善速度的平方。计算机设计者应据此做出相应的规划。

晶体管性能与连线的规模

集成电路的加工工艺是用特征尺寸来表示的,特征尺寸是晶体管或者连线在 x 或 y 方向上的最小尺寸。从1971年到2006年,特征尺寸从10微米降到了0.09微米;实际上我们已转用新的转换单位,称2006年的产品工艺为90纳米,并且65纳米的芯片也正在开发之中。由于芯片上每平方毫米晶体管的数目由单个晶体管的表面积大小决定,所以晶体管的密度与特征尺寸大小的平方成反比。

而晶体管性能的提高更为复杂,当特征尺寸减小时,器件在水平方向以特征尺寸线性减小速度的平方减小,而且在垂直方向上也减小。在垂直方向上尺寸的减小需要操作电压同时降低,以保证操作的正确和晶体管的可靠性。上述因素的结合导致了晶体管性能和工艺特征尺寸之间的复杂关系。一般而言,晶体管性能随着特征尺寸的减小线性增加。

随着特征尺寸缩小,晶体管数目按平方增加而晶体管的性能线性改善这一事实对计算机系统结构来说既是挑战也是机会,所以才会产生计算机系统结构师。在微处理器问世之初,晶体管密度的高速增加使得处理器从4位发展到8位、16位和32位。最近,晶体管密度的增加又推动了64位微处理器的产生及与流水线及Cache相关的诸多创新,这些内容将在第2章、第3章、第5章介绍。

尽管晶体管的性能会随着特征尺寸的减小而增加,但集成电路中连线的情况却与此不同,特别是连线的信号时延,与电阻和电容的乘积成正比。当然,特征尺寸减小时连线也会相应缩短,但是单位长度上的电阻和电容却相应增加。这个关系比较复杂,因为电阻和电容取决于具体的加工方法、连线的几何形状、连线的负载甚至和其他结构的连接。有时工艺会带来改进,比如用铜作为连接介质,这将会减少一定的连线时延。

总体而言,与晶体管性能的改进相比,连线延迟的改进空间更大,这为设计者带来了更多的机会。在过去的几年里,连线时延已经成为大规模集成电路设计中的主要障碍,而且往往比晶体管开关时延更为关键。越来越多的时钟周期消耗在信号传输的连线时延上。2001年,Intel Pentium 4在其20级左右的流水线中专门分配了两级流水线用于芯片上的信号传输。

1.5 集成电路功耗的发展趋势

当器件升级时,功耗同样提出了挑战。第一,现代微处理器使用大量的管脚和多层互连结构以满足芯片功率和接地的需求;第二,芯片功率会产生热量,应采取适当的散热措施。

对于CMOS芯片来说,最主要的能耗通常来自开关晶体管(switching transistor),也称为动态功率。一个晶体管需要的功率($\text{Power}_{\text{dynamic}}$)与以下参数的乘积成比例,这些参数包括晶体管的电容性负载(Capacitive load)、电压(Voltage)的平方以及开关频率(Frequency switched),计量单位是瓦特:

$$\text{Power}_{\text{dynamic}} = 1/2 \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}$$

相对于功率,移动设备更加关注电池寿命,因此适宜用焦耳作为能量($\text{Energy}_{\text{dynamic}}$)的度量单位:

$$\text{Energy}_{\text{dynamic}} = \text{Capacitive load} \times \text{Voltage}^2$$

由此,通过降低电压可以在很大程度上减少动态功率和能耗,因此,在过去的20年中,电压从5V降到了接近1V的水平。电容性负载是由连接到输出的晶体管数目以及决定电线和晶体管

电容的实现技术所作用的。对于一个确定的任务来说,减慢时钟频率只能减少功率,而不能减少能耗。

例题 目前一些微处理器采用了可调整电压技术,当电压下降 15% 时,频率也会下降 15%。这对动态功率有什么影响?

解答: 由于电容不变,电压和频率之间的比率为

$$\frac{\text{Power}_{\text{new}}}{\text{Power}_{\text{old}}} = \frac{(\text{Voltage} \times 0.85)^2 \times (\text{Frequency switched} \times 0.85)}{\text{Voltage}^2 \times \text{Frequency switched}} = 0.85^3 = 0.61$$

因此,将会比原来情况减少 60% 的功率。

在处理器发展的过程中,开关晶体管的数目以及其开关频率的增长要比负载电容和电压的减少更为重要,因此会导致功率和能耗总体上的增加。第一个微处理器只消耗 0.1 W 的功率,而到了最新版的 3.2 GHz Intel Pentium 4 则需要消耗 135 W 的功率。另外,在边长大约 1 cm 的芯片上要及时散热,但目前我们已经接近了风冷技术的极限。一些 Intel 微处理器使用了热敏二极管,每当芯片过热时,热敏二极管会自动降低工作负载,例如,它们可以降低电压和时钟频率或指令发射速率。

合理分配功率、散热以及避免过热点在技术上已经变得越来越困难。现在电源问题已经成为使用晶体管时的瓶颈;在过去,这种限制主要在原始的硅区域。由于电源的限制,目前绝大多数微处理器通过采用关掉时钟或者不工作的组件来节省能量和动态功率。例如,如果没有浮点指令在执行,可以中止浮点装置的时钟。

虽然动态功率是 CMOS 功率消耗的主要来源,静态功率 ($\text{Power}_{\text{static}}$) 也是一个重要的问题,因为晶体管停止工作时仍然会有泄漏电流 ($\text{Current}_{\text{static}}$):

$$\text{Power}_{\text{static}} = \text{Current}_{\text{static}} \times \text{Voltage}$$

因此,即使晶体管被关掉不用,增加其数量也会增加功率,而且在处理器中,晶体管越小泄漏电流越大。这样造成的结果是,极低功率的系统甚至给不工作组件提供一定的电压以控制泄漏造成的能耗。2006 年,在泄漏控制方面的目标是不超过总功率消耗的 25%,但是在高性能系统设计中常常会远远超过这一目标。就像前面提到的,为了克服风冷技术的限制,设计者们正在探索更低电压和时钟频率的单芯片多处理器技术。

1.6 成本的发展趋势

尽管在有些种类计算机设计(特别是超级计算机的设计)中成本因素可以不予考虑,但是实际上注重成本的设计已经变得越来越重要。的确,在过去的 20 年中,计算机行业主要是靠改进技术来提高性能和降低成本的。

为了不因成本的变化使得教科书内容过时,也由于成本问题很复杂,在不同工业部门表现不一样,所以教科书中往往忽略性价比中的成本因素。但是,当设计者想增加某一功能特性时,就必须考虑成本要素,否则就会像建筑师在对钢筋和水泥的行情一无所知的情况下就去设计摩天大楼一样不可想象。

这一节讨论影响成本的主要因素和发展趋势,以及它们随时间变化的情况。

时间、产量、产品化的影响

即使计算机的底层实现技术没有实质性的改进,计算机元件的生产成本也会逐渐降低,这是由于学习曲线——制造成本会随时间而降低在起作用。学习曲线本身最适合用产出率的变化来衡量,产出率就是合格产品占总量的百分比。无论生产的产品是芯片、主板还是一个系统,能使产出率加倍的设计,就能降低一半成本。

深入理解学习曲线对产出率的改善是在产品生命周期中控制成本的关键。比如,每兆字节 DRAM 的价格长期以来就以每年 40% 的速度下降。因为 DRAM 芯片的价格趋势与其成本紧密联系在一起——除去产品缺货和过剩时期,DRAM 的价格和成本基本成正比。

微处理器的价格同样在下降,但是由于它没有 DRAM 芯片那么标准,所以它的价格与成本之间的关系就显得更加复杂。在竞争激烈时期,价格走向和成本很接近,但微处理器生产商很少会低于成本出售产品。图 1.10 展示了 Intel 微处理器的价格变化趋势。

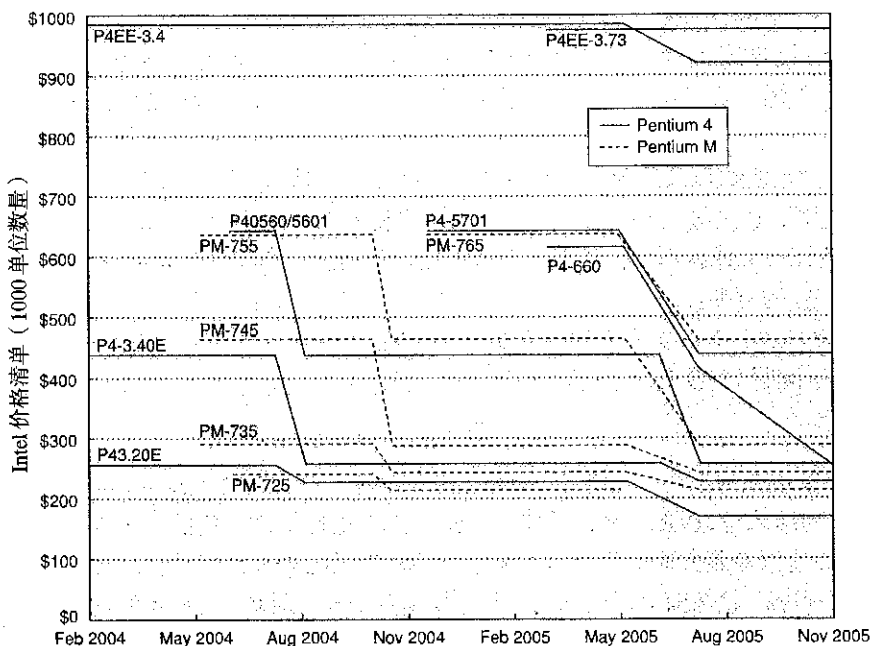


图 1.10 当成品率提高和竞争压力共同导致成本降低时,特定主频的 Intel Pentium 4 和 M 处理器的价格随着时间的推移在以一定的速率下降。最新的数据表明 Pentium 处理器的价格将持续下跌直到与当前的最低成本部件相近(200 美元),并呈现出一种价格与成本密切相关并同时下降的局面。数据来自于 2005 年 5 月的微处理器报告

除此之外,产量也是决定成本的重要因素。产量会以如下几种方式影响成本。其一,产量增加会缩短达到学习曲线中最优生产效率的时间,这一时间某种程度上与系统或芯片的产量成正比;其二,产量增加可以提高采购和生产的效率,从而降低成本。根据以往的经验,一些设计者曾经估计如果产量每增加一倍,成本就会降低 10%;其三,随着产量的增加,每件产品平均的开发费用会降低,因而产品的成本和售价就会越来越接近。

商品是由多家厂商大量出售的本质相同的产品。事实上,零售店出售的所有产品都可以成为商品,就像标准的 DRAM、磁盘、显示器和键盘等。在过去 15 年中,大部分低端计算机产业已变成了制造运行微软 Windows 的台式机和笔记本电脑的商品产业。

因为有多家厂商生产实际上相同的产品,因而在该领域竞争就会异常激烈。当然,这种竞争减小了售价与成本间的差价,同时也降低了成本。这是因为零售市场需求量大,且产品有明确的规范,这使得许多厂家可以参与到生产零配件的竞争中来。零配件供应商之间的激烈竞争和零配件产量增加带来的高生产率最终将导致零配件总生产成本的下降。尽管只有非常有限的利润(这也是所有商品行业的特征),但这也使计算机行业在低端产品方面可以获得更高的性价比和更大的产量。

集成电路的成本

为什么一本计算机系统结构的教科书会有一节专门来讨论集成电路的成本呢?在竞争日益激烈的计算机市场中,标准部件——如磁盘、DRAM等——在计算机总成本中占的比例越来越高,因而集成电路的成本成了影响计算机总成本的重要因素,在产量大、注重成本的市场中尤其如此。因此,要想分析整机的成本就必须先研究芯片的成本。

尽管集成电路的成本呈指数下降趋势,但是生产硅片的基本程序并没有改变:先对一个晶圆进行测试,然后把晶圆切分为晶片,再对每个晶片进行封装(如图1.11和图1.12所示)。因此,一片封装好的集成电路的成本为

$$\text{集成电路的成本} = \frac{\text{晶片成本} + \text{晶片测试成本} + \text{封装成本}}{\text{最终成品数目}}$$

本节中我们集中讨论晶片的成本,并在本节末尾概括介绍影响测试和封装成本的主要因素。

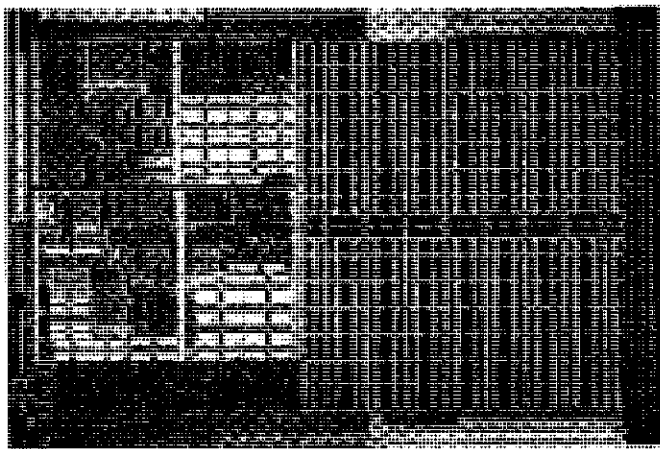


图 1.11 AMD Opteron 微处理器晶片图 (AMD 提供)

要想知道每个晶圆上有多少性能良好的晶片,首先要知道晶圆上的晶版个数和合格晶片所占的比例。由此可以很简单地推算晶片的成本:

$$\text{晶片成本} = \frac{\text{晶圆成本}}{\text{每片晶圆的晶片数} \times \text{晶片的成品率}}$$

有趣的是,这个晶片的成本公式表明晶片的成本对晶片的大小非常敏感,下面将说明这一点。

每个晶圆上晶片的个数近似为晶圆面积除以晶片面积,更确切的公式为

$$\text{每片晶圆的晶片数} = \frac{\pi \times (\text{晶圆的直径}/2)^2}{\text{晶片的面积}} - \frac{\pi \times \text{晶圆的直径}}{\sqrt{2} \times \text{晶片的面积}}$$

第一项是晶圆面积 (πr^2) 与晶片面积的比, 第二项是对晶圆边缘附近的方形晶片数目的修正。晶圆的周长 (πd) 除以正方形晶片的对角线长度近似为晶圆边缘处晶片的数目。

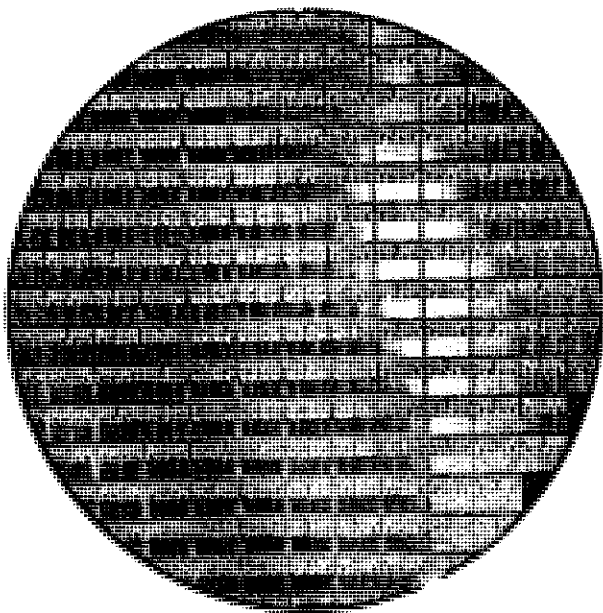


图 1.12 包含 117 个 AMD Opteron 芯片 (在 90 nm 处理器上实现) 的 300 毫米晶圆图 (AMD 提供)

例题 直径为 300 mm (30 cm) 的晶圆上有多少边长为 1.5 cm 的晶片?

解答: 晶片的面积为 2.25 cm^2 。所以

$$\text{每片晶圆的晶片数} = \frac{\pi \times (30/2)^2}{2.25} - \frac{\pi \times 30}{\sqrt{2} \times 2.25} = \frac{706.9}{2.25} - \frac{94.2}{2.12} = 270$$

但这仅给出了每个晶圆上可以容纳的晶片的最大数目, 关键的问题是每个晶圆上合格晶片的比例, 即晶片的成品率是多少。集成电路成品率的一个简单模型假定有缺陷的产品在晶片中随机分布且成品率与制造过程的复杂度成反比, 所以有如下的公式:

$$\text{晶片的成品率} = \text{晶圆的成品率} \times \left(1 + \frac{\text{单位面积内的残次品数目} \times \text{晶片面积}}{\alpha} \right)^{-\alpha}$$

这个公式是通过统计大量生产线而得到的经验模型。其中晶圆的成品率除去了完全被损坏因而不必进行测试的晶圆。为简单起见, 我们假定晶圆成品率为 100%。单位面积的残次品数目用来衡量随机的制造缺陷。在 2006 年, 根据工艺成熟程度 (回忆一下前面提到过的学习曲线), 对于 90 nm 的标准来说, 这一指标约为 $0.4/\text{cm}^2$ 。最后, α 是一个衡量工艺复杂程度的参数, 大约与掩膜层数相对应。对于 2006 年的多层金属 CMOS 生产流程而言, α 取 4.0 较好。

例题 设单位面积残次品密度为 $0.4/\text{cm}^2$, 且 $\alpha = 4.0$, 分别求边长为 1.5 cm 和 1.0 cm 的晶片的成品率。

解答: 晶片的面积分别为 2.25 cm^2 和 1 cm^2 , 面积较大的晶片的成品率为

王。晶圆

$$\text{晶片的成品率} = \left(1 + \frac{0.4 \times 2.25}{4.0}\right)^{-4} = 0.44$$

面积较小的晶片的成品率为

$$\text{晶片的成品率} = \left(1 + \frac{0.4 \times 1.00}{4.0}\right)^{-4} = 0.68$$

即面积较大的晶片有不到一半是合格的，而面积较小的晶片有超过 2/3 是合格的。

用每个晶圆上的晶片数目乘以晶圆的成品率(已经考虑了残次品)就得到每个晶圆上合格晶片的数目。由上面的例子可以估算出直径为 300 mm 的晶片上有 120 个面积为 2.25 cm² 的合格晶片，或 435 个面积为 1 cm² 的合格晶片。在现代 90 nm 的工艺下，大多数 32 位和 64 位微处理器的晶片尺寸都在这两个尺寸之间。低端嵌入式 32 位处理器有时面积小到 0.25 cm²，用于打印机、汽车等领域的嵌入式微处理器有时还不到 0.1 cm²。

在商品化产品(如 DRAM 和 SRAM)的巨大价格压力之下，设计者不得不采用冗余作为一种提高成品率的手段。很多年来 DRAM 常常通过多包含一些存储单元以防出现缺陷。在标准 SRAM 和微处理器的 Cache 使用的更大的 SRAM 阵列中，设计者们已经采取了类似的技术。显然，这种冗余方法的出现对提高成品率是很有效的。

在 2006 年，使用最先进的技术处理一个直径为 300 mm (12 英寸)的晶圆需要花费 5000~6000 美元的成本。假设一个处理过的晶圆成本是 5500 美元，那么 1 个 1.00 cm² 的晶片的成本将大约为 13 美元，而 2.25 cm² 的晶片的成本将大约为 46 美元，也就是说，假如某块晶片面积是另一块晶片的两倍，那么成本就大约是它的 4 倍。

提供)

对于芯片的成本，计算机设计者应该考虑哪些方面呢？生产流程决定了晶片的成本、晶片的成品率和单位面积的残次品数目，因此，设计者唯一能控制的就是晶片的面积。实际上，由于单位面积的残次品数目不大，因此，每个晶圆上好的晶片的数量以及每块晶片的成本增长的速度与晶片面积增长速度的平方成正比。计算机设计者可以通过确定晶片所包含的功能特性及 I/O 管脚数目来影响晶片的大小，从而影响晶片的成本。

在决定把一个部件用于计算机前，必须对它进行测试(区分合格与不合格晶片)、封装和再测试。显然，这些过程都会增加成本。

晶片的比
直机分布

上述内容重点分析了生产一个合格晶片的可变成本，这非常适宜集成电路的大批量生产。然而对于那些小批量(例如产量不超过 100 万)的集成电路而言，掩膜组的成本是一个非常重要的固定成本。集成电路处理过程的每一步都需要一个单独的掩膜。所以，对于现代 4 到 6 层金属层的高密集度工艺制造过程来说，掩膜成本通常超过 100 万美元。显然，这笔不小的固定成本开支影响着制造原型和调试运行，从而成为小批量生产成本的一个重要部分。由于掩膜的成本价格有可能继续上涨，所以设计者们可能采用可重新配置的逻辑组件来提高某个部件的灵活性，或者选取一些门阵列(它们的定制掩膜少一些)，从而降低掩膜的成本。

破坏因
目用来
)，对于
与掩膜

成本与价格

随着计算机商品化程度的提高，产品的成本和产品销售的价格差距已经在逐渐缩小。这种差价是为了填补公司的研发费用、营销费用、设备维护费用、场地租金、财务成本、税前利润和赋税。许多工程师对大多数公司的研发费用仅占其收入的 4% (PC 业务) 到 12% (高端服务器业务) 感到意外，而且这笔费用包括所有的工程研发费用。

晶片的成
本

1.7 可靠性

从历史经验上来看,集成电路是计算机中最可靠的组件之一。虽然它们的管脚容易损坏,且在通信信道中常常会发生故障,但在芯片内部的差错率很低。当采用 65 nm 或者更小的制造工艺时,瞬时和永久的故障都会变得十分普遍。这就需要所设计的计算机系统能够应对这些新的挑战。本节仅对可靠性问题做一个简单回顾,而把有关术语和方法的权威定义将在 6.3 节中介绍。

为了更好地设计和构建计算机,通常把它分为不同的抽象层次。将计算机看成由许多子系统组成的,可以逐层分解细化直至单个晶体管。虽然某些故障常常出现,但大多数可以被局限在模块的单个组件上。因此,一个层面上某个模块的完全失效,反映在高层模块上仅仅被认为是组件的误差。这种层次的划分对于构建可靠的计算机是十分有益的。

确定系统是否正常运行是一个难解的问题。在 Internet 服务普及的背景下,这个问题变得非常具体。Internet 服务提供商 (ISP) 采用服务等级协议 (SLA, Service Level Agreements) 或者服务等级目标 (SLO, Service Level Objectives) 来确保网络和电源服务的可靠性。例如,如果服务提供商每月超过一定时间没有兑现对消费者的承诺,他们就要支付相应的赔偿。因此,服务等级协议 (SLA) 可以用来决定系统何时工作或停止。

关于 SLA, 系统可以在两种服务状态中做出选择:

1. 服务实现: 交付的服务与 SLA 相符。
2. 服务中断: 交付的服务与 SLA 不符。

在这两种状态间的转换可以被失效 (由 1 状态到 2 状态) 或恢复 (由 2 状态到 1 状态) 触发。对这些转换的量化研究,得到了可靠性度量的两个主要方法:

- 模块可靠性是从模块可用直至发生故障时的持续服务实现的度量。因此,平均故障时间 (MTTF, Mean Time To Failure) 是一个可靠性度量方法。MTTF 的倒数是故障率,它一般以每 10 亿小时运行中的故障时间来计算,或称为 FIT (Failure In Time)。因此,1 000 000 小时 MTTF 等于 1000 FIT。服务中断以平均修复时间 (MTTR, Mean Time To Repair) 来度量。平均无故障时间 (MTBF, Mean Time Between Failure) 为平均故障时间和平均修复时间相加之和。虽然 MTBF 被广泛使用,但实际上,有时 MTTF 是一个更实用的指标。如果一组模块的生存周期呈指数分布——即单个模块的使用期对于这组模块的故障率来说并不重要——则这组模块的总故障率就是各个模块故障率之和。
- 模块可用性是对于在完成和中断两个状态间变迁下所完成的服务度量。对于需要修复的非冗余系统来说,模块可用性为

$$\text{模块可用性} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})}$$

目前,可靠性和可用性已经可以用量化方法进行分析。使用上述定义,在确定组件可靠性和故障独立性后,就可以用量化的方法来评估一个系统的可靠性。

例题 假设一个磁盘子系统有如下组件和 MTTF:

- 10 个磁盘, 每一个的 MTTF 是 1 000 000 小时的 MTTF
- 1 个 SCSI 控制器, 500 000 小时的 MTTF
- 1 个电源, 200 000 小时的 MTTF

- 1 个风扇, 200 000 小时的 MTTF
- 1 条 SCSI 电缆, 1 000 000 小时的 MTTF

假设生存周期是按指数分布的, 并且故障具有独立性, 将系统的 MTTF 作为一个整体来计算。

解答: 总的故障率为

$$\begin{aligned}\text{系统故障率} &= 10 \times \frac{1}{1\,000\,000} + \frac{1}{500\,000} + \frac{1}{200\,000} + \frac{1}{200\,000} + \frac{1}{1\,000\,000} \\ &= \frac{10 + 2 + 5 + 5 + 1}{1\,000\,000 \text{ 小时}} = \frac{23}{1\,000\,000} = \frac{23\,000}{1\,000\,000\,000 \text{ 小时}}\end{aligned}$$

或 23 000 FIT。这个系统的 MTTF 是故障率的倒数:

$$\text{系统的 MTTF} = \frac{1}{\text{系统故障率}} = \frac{1\,000\,000\,000 \text{ 小时}}{23\,000} = 43\,500 \text{ 小时}$$

即差不多 5 年。

应对故障的主要方法是冗余, 或者是时间冗余 (利用重复操作判断是不是仍有问题), 或者是资源冗余 (用其他组件接管故障的部分)。一旦故障模块被替换, 系统将被完全修复, 系统的可靠性就完好如初了。下面的例子将用量化方法分析冗余带来的好处。

例题 磁盘子系统常常配置有备份电源以提高系统可靠性。使用上一例题中所列组件及其 MTTF, 计算备份电源的可靠性。假设一个电源足以有效支持该磁盘子系统, 此外再配置一个备份电源。

解答: 我们需要一个公式来说明当我们能容忍一个故障并继续提供服务时的情况。为了简化运算, 假定模块的生存周期是按指数分布的, 并且模块间的故障具有独立性。我们的冗余电源系统的 MTTF 是至第一个电源发生故障的平均时间除以第一个电源被没有修复而第二个电源又发生故障的概率。因此, 如果第二个电源在第一个电源被修复前发生故障的概率很小, 则这对电源的 MTTF 就会很大。

由于拥有两个电源, 其故障也没有相关性, 因此到一个磁盘故障的平均时间即为电源 MTTF/2。第二个电源故障的概率接近于 MTTR 除以到另一个电源故障时的平均时间。由此, 这对电源 MTTF 的近似值为

$$\text{电源对 MTTF} = \frac{\text{MTTF}_{\text{power supply}}/2}{\frac{\text{MTTR}_{\text{power supply}}}{\text{MTTF}_{\text{power supply}}}} = \frac{\text{MTTF}_{\text{power supply}}^2/2}{\text{MTTR}_{\text{power supply}}} = \frac{\text{MTTF}_{\text{power supply}}^2}{2 \times \text{MTTR}_{\text{power supply}}}$$

使用上面得到的 MTTF 值, 如果假定 24 小时一直有操作人员在监控电源, 一旦发生故障就立刻替换电源, 那么这对备份电源的可靠性为

$$\text{电源对 MTTF} = \frac{\text{MTTF}_{\text{power supply}}^2}{2 \times \text{MTTR}_{\text{power supply}}} = \frac{200\,000^2}{2 \times 24} \approx 830\,000\,000$$

所以这对电源要比单一电源可靠 4150 倍。

以上对成本、电源和计算机实现技术的可靠性进行了量化分析, 下面将对性能进行量化分析。

1.8 测量、报告和总结计算机的性能

一台机器比另一台机器快的确切含义是什么呢? 对于计算机用户来说, 当同一个程序在某台机器上运行的时间较短时, 用户就认为这台计算机比较快。但是对于 Amazon 网站的管理人员来说, 在单位时间内处理交易多的机器比较快。计算机用户关心的是如何减小响应时间 (也称为执行时间), 即一个事件从开始到结束所用的时间。而一个大的数据处理中心的管理人员关心的是如何增大吞吐量——单位时间内完成的工作总量。

在比较各设计方案时, 我们经常希望比较两台机器 (X 和 Y) 的性能。在这里, “X 比 Y 快” 指的是对于给定的任务, X 的响应时间或执行时间比 Y 的短。而 “X 的速度是 Y 的 n 倍” 是指

$$\frac{\text{Y 的执行时间}}{\text{X 的执行时间}} = n$$

由于执行时间与性能成反比, 所以下述关系成立:

$$n = \frac{\text{Y 的执行时间}}{\text{X 的执行时间}} = \frac{\frac{1}{\text{Y 的性能}}}{\frac{1}{\text{X 的性能}}} = \frac{\text{X 的性能}}{\text{Y 的性能}}$$

“X 的速度是 Y 的 1.3 倍” 是指单位时间内 X 机器完成的任务是 Y 机器完成任务数量的 1.3 倍。

遗憾的是, 人们在比较机器性能时并不总是用时间做标准。通常认为有效而可靠度量性能的指标就是实际程序的执行时间。任何不以时间做标准或不用实际程序进行测试的方法都会产生误导, 甚至导致计算机设计的失误。

即便是执行时间也可以根据需要进行不同的定义。时间的最直接定义也称为墙钟时间、响应时间或消逝时间, 指的是完成一项任务所需的时延, 包括磁盘访问时间、存储器访问时间、输入输出操作时间、操作系统开销——总之, 所有任务的执行时间都包括在内。在多个程序同时运行的计算机中, 当一个程序等待 I/O 操作完成时处理器会转而执行另一个程序, 此时计算机不一定会使每个程序的消逝时间最短。这样, 我们就需要另一个术语来描述这种情况。CPU 时间体现了这一区别, 因为它指的是 CPU 计算耗用的时间, 而不包括等待 I/O 操作完成或运行其他程序的时间 (很明显, 用户感受到的响应时间是程序从开始执行到结束全过程所用的时间, 而不是 CPU 时间)。

经常使用相同程序的用户最有资格对一台新机器的性能进行评价。为测试新系统的性能, 只需比较相同负载的执行时间——程序和所用操作系统指令的总执行时间——即可。遗憾的是, 这种情况形少而又少。大多数情况下, 用户必须依靠其他方法和评测程序来评价机器的性能, 并希望这些方法能够预测新机器的性能。

基准测试程序

测量性能的最好的基准测试程序是真实的应用, 比如编译器等。运行比实际应用简单的程序来评价性能缺乏说服力, 这包括:

- 程序内核: 实际程序中的小而关键的部分。
- 小型基准测试程序: 程序只有 100 行代码, 如 quicksort。
- 综合基准测试程序: 模拟实际应用的特征和行为而编写的程序, 如 Dhrystone。

目前,以上三种方法都缺少可信度,因为编译器的开发人员和系统结构设计者可以共同使计算机在运行这些标准程序时比实际程序要快得多。

另外一个问题是基准测试程序的运行条件。改善基准测试程序性能的一种方法是采用基准测试程序专用标记,但这些标记常常会在许多程序中引起不合法的转换过程或者降低其他程序的性能。为了限制这种情况的出现以及提高结果的可信度,基准测试程序的设计者常常要求生产商对使用同一语言(C或FORTRAN)的所有程序使用同一个编译器和同一套标记。除了编译器标记,还有另外一个问题是是否允许修改源码,解决这个问题有以下3种不同的方案:

1. 不允许修改源码。
2. 允许修改源码,但实际上不可行。比如,数据库基准测试程序依赖于标准数据库程序,约有几千万行代码。这样数据库公司就不可能为了改善某种特定计算机的性能而对源码做出修改。
3. 允许修改源码,只要修改后的版本得到相同的输出即可。

至于是否允许修改源码,基准测试程序设计者需要综合考虑如下因素:这样的修改是否会影响实际的执行效果?是否能为用户提供有用的经验?是否会减小基准测试程序的准确性?

为了克服上述问题,常常把一些基准测试应用集中起来(称为基准测试程序集)来有效评测处理器处理各种应用的性能。当然,这种成套的基准测试程序与单个基准测试程序的功能差不多,但它最大的优点是某种基准测试程序的不足可以被其他基准测试程序所弥补。基准测试程序套件的目标是比较两台计算机的相关性能,尤其是在运行不属于套件的程序的情况下。

这里给出一个典型的例子,EDN嵌入式微处理器基准测试程序协议就是41个用来评估不同嵌入式应用性能的程序内核集,这些应用来自于自动化/工业化、消费者、网络、办公室自动化以及电信领域。由于EEMBC使用的是程序内核,再加上它在性能报告上的选择性,导致了它不能作为一种能有效覆盖不同嵌入式计算机相关性能的预测器。而其一直想取代的综合基准测试程序Dhrystone,至今仍然在一些嵌入式应用中使用。

SPEC是最成功的基准测试程序集之一,它是在20世纪80年代末期为了评估工作站性能而推出的。随着计算机产业的发展,需要多种面向不同应用的基准测试程序集。目前SPEC及其衍生的基准测试程序集基本覆盖了各种不同的应用领域。全部的SPEC基准测试程序集与测试报告都可以在网站 www.spec.org 上找到。

在接下来的许多章节中我们将着重讨论SPEC基准测试程序集,但目前已经有很多公司正在开发新的基准测试程序集,并用来对基于Windows操作系统的PC的性能进行评估。

桌面基准测试程序

桌面基准测试程序分为两大类:面向处理器的和面向图形系统的基准测试程序,大多数面向图形系统的基准测试程序也可以完成面向处理器的功能。最初针对处理器性能测试开发出的SPEC基准测试程序集称为SPEC89,到现在已经发展到第5版——SPEC CPU2006,期间的版本还有SPEC2000, SPEC95, SPEC92和SPEC89。SPEC CPU2006包括12个定点基准测试程序(CINT2000)和17个浮点基准测试程序(CFP2000)。图1.13给出了SPEC CPU2006及其前几个版本的有关信息。

SPEC基准测试程序是经过精简并将I/O对性能的影响减到最小的应用程序。定点基准测试程序的差异很大,范围从C编译器的一部分到“下棋”程序,再到量子计算机模拟程序。浮点基准测试程序包括有限元模型的结构网格代码、关于分子动力学的粒子程序代码和流体动力学的稀疏线性代数代码。SPEC CPU基准测试程序对桌面系统性能和单处理器服务器性能的测试均有价值。在本书中,我们将能看到很多使用这些程序测得的数据。

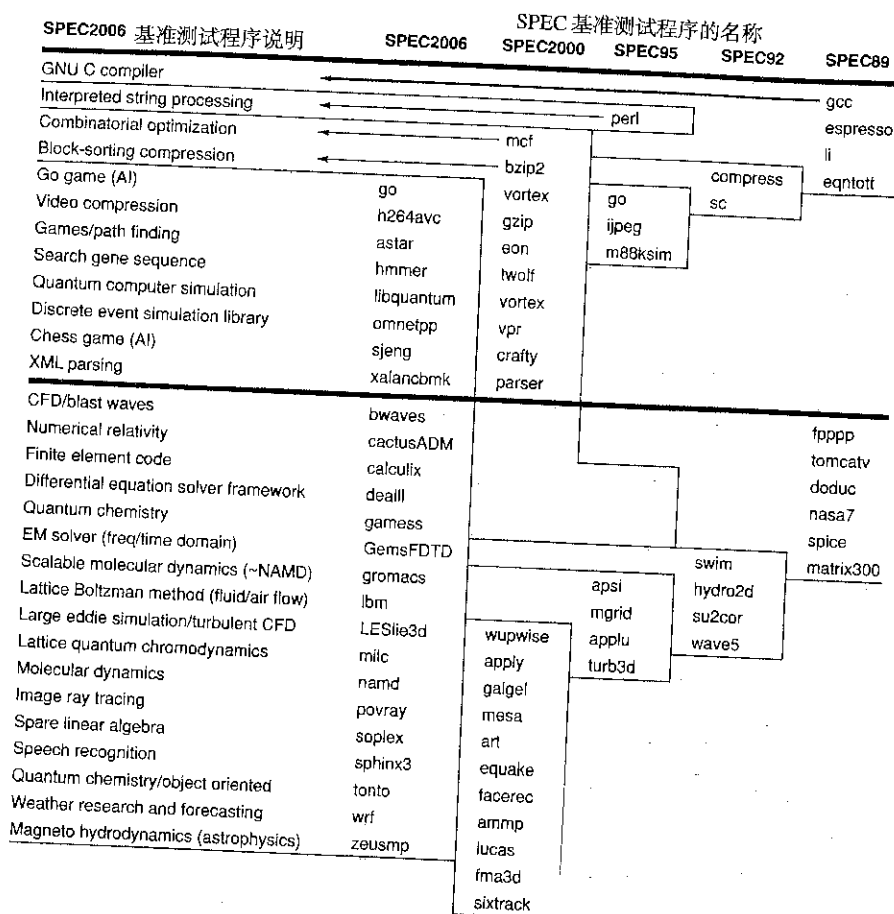


图 1.13 SPEC2006 程序 SPEC 基准测试程序集的发展历程，上半部分表示定点程序，下半部分表示浮点程序。在 12 个 SPEC2006 定点程序中，9 个用 C 开发，其余的用 C++ 开发；浮点程序中，6 个用 FORTRAN 开发的，4 个用 C++ 开发，3 个用 C 开发，还有 4 个用 FORTRAN 和 C 混合开发。图中给出了在 1989，1992，1995，2000 和 2006 几个版本中的全部 70 个程序，最左边的是对 SPEC2006 独有的基准测试程序的描述。同一行中不同版本的 SPEC 程序一般是没有联系的。例如，fpppp 不是 CFD 代码，这和 bwaves 不一样。Gcc 是程序集中使用时间最长的程序。只有 3 个定点程序和 3 个浮点程序出现在三个或三个以上的版本中。在 SPEC2006 中所有的浮点程序都是新开发的。虽然不同时期的 SPEC 中都使用了某些名称相同的程序，但程序版本却不同，并且输入参数或程序大小经常发生改变，以提高 CPU 有效运行时间、避免测量的扰动、防止非 CPU 时间占据大部分运行时间

在 1.11 节中，我们将介绍一些在 SPEC 基准测试程序发展历程中出现的问题，以及在维护可用、高效的基准测试程序过程中所面临的挑战。虽然 SPEC CPU2006 是面向处理器性能评价的，但它同样也适用于面向图形和 Java 程序的基准测试。

服务器基准测试程序

由于服务器功能的多样性，因此也有多种基准测试程序用于其性能评价。其中，最简单的是仅面向处理器处理能力的基准测试程序。SPEC CPU2000 使用 SPEC CPU 构建了一个简易的处理能力

基准程序, 这个程序能够运行 SPEC CPU 基准测试程序的多份复制程序 (一般和处理器数目相同), 并将 CPU 时间转换成处理器的处理比率。这种测试方法称为 SPECrate。

与 SPECrate 不同的是, 由于磁盘和网络访问会导致大量 I/O 操作, 所以大部分服务器应用程序和基准程序应包括面向文件服务器系统、网络服务器、数据库系统以及事务处理系统的基准测试程序。SPEC 提供文件服务器基准程序 (SPECIFS) 和网络服务器基准程序 (SPECWeb)。SPECIFS 是一个通过使用文件服务器请求脚本文件来测量 NFS (网络文件系统) 性能的基准程序, 它能够测量 I/O 系统 (包括磁盘和网络 I/O) 性能和处理器性能。SPECIFS 是一个面向吞吐量、但对响应时间有严格要求的基准程序 (第 6 章将详细讨论一些文件和 I/O 系统基准程序)。SPECWeb 是一个网络服务器基准测试程序, 它能够模拟多个客户端对静态和动态网页的请求, 以及模拟客户端向服务器发布数据的操作。

事务处理 (TP) 基准测试程序用于评测一个系统处理事务的能力, 事务处理包括数据库访问和更新。机票预订系统以及银行的 ATM 系统是典型的简单 TP 系统。更复杂的 TP 系统包括复杂的数据库和决策系统。20 世纪 80 年代中期, 一些工程师组成了独立于厂商的事务处理委员会 (TPC), 来构建客观的 TP 基准测试程序。TPC 基准测试程序在网站 www.tpc.org 上有详细的描述。

第一个 TPC 基准测试程序 TPC-A 于 1985 年发布, 此后, 先后被另外几个更强的基准测试程序所取代。TPC-C 于 1992 年开始创建, 它能够模拟复杂的查询应用场景。TPC-H 模型主要面向自主的决策支持——查询具有无关性, 先前查询的信息不能用来优化后续的查询。TPC-W 是一个基于网络的事务处理基准测试程序, 它能够在可控的基于 Internet 的网络环境中执行, 以模拟一个商业应用事务处理网络服务器的情况。最新的是 TPC-App, 它是一个应用服务器和 Web 服务基准测试程序, 可以模拟运行在 24 × 7 环境中的交易处理应用服务器的情况。

所有的基准测试程序都以每秒钟处理的事务数来评测系统的性能。此外, 它们对响应时间也有要求, 当响应时间达到极限值时测量吞吐量。用于模拟真实世界的大型系统主要表现在用户数量和数据库规模上, 它通常具有较高的事务处理率。最后还要考虑运行基准测试程序系统的性能成本, 这样才能更准确地比较性价比。

性能评价报告

性能评价报告的指导原则是**可重现性**: 列出其他研究人员若想重现该实验所需要的所有信息和资料。一份 SPEC 基准测试报告需要对机器、编译器标志进行非常详尽的说明, 而且应当公布基准性能和优化结果。除硬件、软件和基准调整参数的设置外, SPEC 报告还应当以表格和图示的方式给出实际性能数据。而 TPC 的基准测试报告更加复杂, 因为它必须包括基准测试程序日志和成本信息。因为厂商在高性能和高性价比方面的竞争十分激烈, 因此, 这些报告对他们来说是评估计算系统实际成本的很好的数据来源。

性能评测结果的总结

在实际的计算机系统中, 设计者必须通过一套参考基准在多个方案中择优选择。同样, 消费者也会依据性能测量基准来选择满足需求的理想计算机。在上述情况下, 需要一套标准的测量方法来评价基准测试程序与重要应用性能之间的相似性, 并使得性能上的差异更加清晰。在理想情况下, 从统计学的角度来看, 基准测试程序集应该能够代表所有应用程序的有效样本。此样本需要更大的基准测试程序数目, 并且使用随机取样方式进行样本选择, 这样才具有典型意义。

一旦选择了一套基准测试程序集来测试性能, 就可以用数字总结出这套基准测试程序集的性能测试结果。分析该结果的一种直接方法就是比较程序执行时间的算术平均值。例如, 在比较算术平

均值的情况下,如果某些 SPEC 程序耗时是其他程序的 4 倍,那么这些程序就更加重要。一种方法是为每项基准测试程序加上一个权值,然后使用这种加权算术平均值作为性能分析的依据。这里的关键在于如何选取权值, SPEC 是由一些有利益冲突的公司联盟推出的,每家公司都有自己所倾向使用的权值集,因此很难达成一致。一种解决方法是让所有的程序在参考计算机上执行相同的时间,但是这会导致参考计算机上的性能特性发生偏差。

比选择权值更好的办法是,可以通过将参考计算机的执行时间除以被测计算机的执行时间这种方法来规范被测计算机的执行时间,得到和被测计算机之间的性能比。这个比率称为 SPECRatio。它与本书中比较计算机性能的方式一致。例如,假定在一个基准测试程序下计算机 A 的 SPECRatio 比另一台计算机 B 的高出 1.25 倍,则可得到

$$1.25 = \frac{\text{SPECRatio}_A}{\text{SPECRatio}_B} = \frac{\frac{\text{参考计算机的执行时间}}{A \text{ 的执行时间}}}{\frac{\text{参考计算机的执行时间}}{B \text{ 的执行时间}}} = \frac{B \text{ 的执行时间}}{A \text{ 的执行时间}} = \frac{A \text{ 的性能}}{B \text{ 的性能}}$$

注意在这里并没有考虑参考计算机的执行时间,而且在量化比较时对参考计算机的选择是不相关的。图 1.14 给出了一个例子。

基准测试程序	Ultra 5 时间 (s)	Opteron 时间 (s)	SPECRatio	Itanium 2 时间 (s)	SPECRatio	Opteron/Itanium 时间 (s)	Itanium/Opteron SPECRatios
wupwise	1600	51.5	31.06	56.1	28.53	0.92	0.92
swim	3100	125.0	24.73	70.7	43.85	1.77	1.77
mgrid	1800	98.0	18.37	65.8	27.36	1.49	1.49
applu	2100	94.0	22.34	50.9	41.25	1.85	1.85
mesa	1400	64.6	21.69	108.0	12.99	0.60	0.60
galgel	2900	86.4	33.57	40.0	72.47	2.16	2.16
art	2600	92.4	28.13	21.0	123.67	4.40	4.40
equake	1300	72.6	17.92	36.3	35.78	2.00	2.00
facerec	1900	73.6	25.80	86.9	21.86	0.85	0.85
ammp	2200	136.0	16.14	132.0	16.63	1.03	1.03
lucas	2000	88.8	22.52	107.0	18.76	0.83	0.83
fma3d	2100	120.0	17.48	131.0	16.09	0.92	0.92
sixtrack	1100	123.0	8.95	68.8	15.99	1.79	1.79
apsi	2600	150.0	17.36	231.0	11.27	0.65	0.65
几何平均值			20.86		27.12	1.30	1.30

图 1.14 Sun Ultra (SPEC2000 的参考计算机) 的 SPECfp2000 执行时间以及 AMD Opteron 和 Intel Itanium 2 的执行时间和 SPECRatio (SPEC2000 将执行时间都乘以 100 以便在结果中不出现小数位,比如 20.86 作为 2086 报告)。最后两列给出了执行时间和各自的 SPECRatio 的比率。本图证明了具有相关性能的参考计算机之间的不相关性。执行时间的比率和 SPECRatio 的比率是相等的,几何平均值的比率和比率的几何平均值也是相等的

由于 SPECRatio 是一个比率而不是绝对的执行时间,所以,必须使用几何平均数计算 (因为 SPECRatio 没有单位,所以使用算术平均是没有意义的)。公式为

$$\text{几何平均数} = \sqrt[n]{\prod_{i=1}^n \text{sample}_i}$$

在SPEC的例子中, sample_i 是程序 i 的 SPECRatio。使用几何平均数保证了如下两个重要特性:

1. 比率的几何平均数和几何平均数的比率一样。
2. 几何平均数的比率和性能比率的几何平均数相等, 这就意味着参考计算机的选择是不相关的。

因此, 使用几何平均数是必要的, 尤其是在使用性能比率来比较计算机时。

例题 证明几何平均数的比率和性能比率的几何平均数相等, 以及参考计算机选择的无关性。

解答: 假设有两台计算机 A 和 B, 每台都有 SPECRatio 集。

$$\begin{aligned} \frac{\text{几何平均数}_A}{\text{几何平均数}_B} &= \frac{\sqrt[n]{\prod_{i=1}^n \text{SPECRatio } A_i}}{\sqrt[n]{\prod_{i=1}^n \text{SPECRatio } B_i}} = \sqrt[n]{\prod_{i=1}^n \frac{\text{SPECRatio } A_i}{\text{SPECRatio } B_i}} \\ &= \sqrt[n]{\prod_{i=1}^n \frac{\frac{\text{参考计算机的执行时间}}{A_i \text{ 的执行时间}}}{\frac{\text{参考计算机的执行时间}}{B_i \text{ 的执行时间}}}} = \sqrt[n]{\prod_{i=1}^n \frac{B_i \text{ 的执行时间}}{A_i \text{ 的执行时间}}} = \sqrt[n]{\prod_{i=1}^n \frac{A_i \text{ 的性能}}{B_i \text{ 的性能}}} \end{aligned}$$

这就证明了 A 和 B 的 SPECRatio 的几何平均数就是在一套基准测试程序中所有基准测试程序下的 A 和 B 的性能比率的几何平均数。图 1.14 展示了使用 SPEC 例子的有效性。

一个平均数能否表示一套基准测试程序的性能是一个关键的问题。如果可以使用标准偏差来描述分布的变化, 就可以确定这个平均数是否有效。更进一步, 如果知道该分布具有一个或几个标准形式, 那么使用标准偏差就更有意义。一种有用的分布是标准正态分布, 其样本数据均衡地分布在平均数周围。另外一种是对数正态分布, 其对数数据 (注意不是原始数据) 是在一个对数范围内正态分布的, 因此在这个范围内也是均衡的 (在线性范围内, 对数正态分布是不均衡的, 因而会在分布图的右侧呈现出一个长长的“尾巴”)。

举例来说, 对两个不同的系统用两种不同的基准测试程序分别进行评价, 如果一个系统在一种基准测试程序下比另外一个快 10X, 而在另一种程序下慢 10X, 则相关性能就是比率集 {1, 10}。但事实上两者具有相同的性能, 即平均数应为 1.0, 实际上它在一个对数范围内是正确的。

为了描述算术平均数的变化性, 我们使用称为 σ 的算术标准偏差 (stdev)。其计算公式为

$$\text{stdev} = \sqrt{\sum_{i=1}^n (\text{sample}_i - \text{平均数})^2}$$

和几何平均数一样, 几何标准偏差是乘法性质而不是加法性质的。使用几何平均数和几何标准偏差, 我们可以简单地获取样本的自然对数, 计算算术平均数和偏差, 并取指数将其转换回来。我们可以用以下公式描述乘法形式的平均数和标准偏差 (gstdev), 通常也称为 σ :

的中器制管用的

器封吸的世

$$\text{几何平均数} = \exp\left(\frac{1}{n} \times \sum_{i=1}^n \ln(\text{sample}_i)\right)$$

$$\text{gstdev} = \exp\left(\sqrt{\frac{\sum_{i=1}^n (\ln(\text{sample}_i) - \ln(\text{几何平均数}))^2}{n}}\right)$$

现代电子表格软件可以提供诸如 EXP() 和 LN() 这样的函数, 这就简化了几何平均数和几何标准偏差的计算。

对于对数正态分布, 我们预测 68% 的样本落在范围 [平均数 / gstdev, 平均数 × gstdev] 内, 95% 的样本落在范围 [平均数 / 2 × gstdev, 平均数 × 2 × gstdev] 内。

例题 使用图 1.14 的数据, 计算几何标准偏差和结果属于几何平均数的单精度标准偏差的比例。结果是否和对数正态分布一致?

解答: 对于 AMD 的 Opteron 处理器, 几何平均数为 20.86; 对于 Intel 公司的 Itanium 2 处理器, 几何平均数为 27.12。从 SPEC Ratios 也可以猜到, Itanium 2 的标准偏差要更高 (1.93 对 1.38) 这表明结果和平均数有很大的不同, 因此预测性较弱。对于 Itanium 2, 单精度标准偏差的范围是 [27.12/1.93, 27.12 × 1.93] 或 [14.06, 52.30]; 对于 Opteron 处理器, 单精度标准偏差的范围是 [20.36/1.38, 20.36 × 1.38] 或 [15.12, 28.36]。对于 Itanium 2, 14 项基准中有 10 项落入一个标准偏差 (71%); 对于 Opteron 处理器, 14 项基准中有 11 项落入一个标准偏差 (78%)。因此, 两个结果都符合对数正态分布。

1.9 计算机设计的量化原则

到现在为止, 我们已经讨论了如何定义、测试、分析计算机的性能、成本、可靠性和电源, 接下来将探讨一些有关计算机设计和分析时的指导性原则。本节介绍了有关计算机设计的重要经验数据, 以及两个用于设计方案评价的重要公式。

采用并行性

采用并行性是改善计算机性能的一种重要方法。本书中的每一个章节都给出了采用并行性增强计算机性能的例子。所给出三个例子的每一个都会后面的有关章节中进行深入分析。

第一个例子涉及系统级并行的使用。为了改善典型服务器基准的吞吐量性能, 比如 SPECWeb 或 TPC-C, 可以使用多处理器和多磁盘技术。处理作业请求的工作量可以在多个处理器和磁盘之间同时进行, 从而实现了更大的吞吐量。这种能够扩展存储器和处理器以及磁盘数目的方式称为可扩展性, 它对于服务器来说具有重要价值。

在单一处理器层面上, 充分利用指令间并行对实现更高性能十分关键。其中最简单的方法之一就是流水线。流水线的基本思想是在一个指令序列中, 让指令重叠执行以减少总的完成时间。这些可以在附录 A 中找到, 另外也是第 2 章关注的焦点。流水线执行的关键在于流水线中并不是每条指令都依赖于它的前一条指令, 这样, 完全或部分地并行执行指令就有可能。关于流水线操作的更多细节可以在附录 A 中找到, 另外, 它也是第 2 章关注的焦点。

此外, 在详细的数字设计层面上也可以发掘并行性。例如, 组相联 Cache 可以使用存储器中的多个存储块, 通过并行查找来找到所需的内容。现代算术逻辑单元 (ALUs) 具有先行进位加法器,

这种加法器能够利用并行性来加速求和的过程,使其与操作数的位数的关系从线性关系变为对数关系。

局部性原理

重要的基本的观察来源于程序的特征。我们经常用到的最重要的程序特征是**局部性原理**:程序经常会重复使用它最近使用过的指令和数据。一个广泛适用的基本原理是,程序花费90%的执行时间来执行10%的代码。局部性原理意味着我们可以利用最近用过的指令和数据在一定误差范围内合理地预测将要用到的指令和数据。局部性原理也可以应用于数据存取,尽管不像在指令存取中效果那么明显。

有两种类型的局部性原理。**时间局部性原理**说明最近访问过的内容很可能即将被再次使用;**空间局部性原理**是说地址邻近的内容可能在一定时间内被连续使用。我们将在第5章讨论这些原理的应用。

关注经常性事件

在计算机设计中,最重要且应用最广泛的准则就是提高经常性事件的执行速度,也就是说,在设计上必须有所取舍时,一定要优先考虑经常性事件。这条原则对如何分配资源同样适用,因为如果事件是经常发生的,则改进的效果会非常明显。

该原则也同样适用于电源的分配。处理器中的取指和译码单元要比乘法单元使用得更加频繁,因此,应优先优化这两个单元。这个原则也适用于可靠性分析。如果一个数据库服务器为每个处理器配备了50个磁盘(如下节),那么如下节所述,这时系统的可靠性主要取决于存储的可靠性。

另外,经常出现的情况一般比不经常出现的情况要简单一些,因而提高性能相对容易。例如,当处理器执行两个数相加的操作时,出现溢出的情况比较少见,不溢出才是比较常见的情况。因此,可以通过优化不溢出相加的操作来提高机器的性能。尽管这样做可能会在溢出时降低机器的速度,但由于溢出是较少出现的情况,所以,总的效果是提高了机器的性能。

本书中会出现许多使用该原则的实例。应用这条原则时,必须首先明确什么是经常性事件,以及提高这种情况下机器的运行速度对计算机的整体性能提高的贡献大小。下面将要讨论的一个相关的基本定律,即**Amdahl定律**,可以将此原则进行量化。

Amdahl定律

通过改进计算机的某一部分,所得到的性能提升可以用**Amdahl定律**定量地反映出来。**Amdahl定律**可以阐述为:通过使用某种较快的执行方式所获得的性能提高,受限于可使用这种较快执行方式的时间所占的比例。

Amdahl定律定义了采用某种增强措施所取得的**加速比**。那么什么是加速比呢?假定如果我们采用了某种增强措施,可以使计算机的性能提高,那么加速比就是下式所定义的比率:

$$\text{加速比} = \frac{\text{使用增强措施时完成整个任务的性能}}{\text{不使用增强措施时完成整个任务的性能}}$$

也可以定义为

$$\text{加速比} = \frac{\text{不使用增强措施时完成整个任务的时间}}{\text{使用增强措施时完成整个任务的时间}}$$

加速比反映了使用增强措施后完成一个任务比不使用增强措施完成同一任务加快的比率。

Amdahl定律为计算某些情况下的加速比提供了一种便捷的方法。加速比主要取决于两个因素:

1. 在原有的计算机上,能被改进并增强的部分在总执行时间中所占的比例。例如,如果一个任务在原来机器上的执行时间为60 s,其中20 s的执行时间可以使用增强措施,那么比率就是20/60。这个值我们称之为增强比例,它总是小于等于1。
2. 通过增强的执行方式所取得的改进,即整个程序使用了增强的执行方式后,该任务执行速度提高的程度。这个值是原来程序的执行时间与使用增强措施后程序的执行时间之比。如果在原来的条件下程序的某一部分执行时间为5 s,而改进后这一部分只需要2 s,那么改进就是5/2。这个值我们称之为增强加速比,它总是大于1的。

机器使用了增强功能后,执行时间等于未改进部分的执行时间加上改进部分的执行时间:

$$\text{新的执行时间} = \text{原来的执行时间} \times \left((1 - \text{增强比例}) + \frac{\text{增强比例}}{\text{增强加速比}} \right)$$

总加速比等于两种执行时间的比:

$$\text{总加速比} = \frac{\text{原来的执行时间}}{\text{新的执行时间}} = \frac{1}{(1 - \text{增强比例}) + \frac{\text{增强比例}}{\text{增强加速比}}}$$

例题 现在我们分析一个用于Web服务器系统处理器的性能。假定采用以下的增强方式,新的处理器处理Web服务器应用程序的运行速度是原来处理器的10倍,同时假定此处理器有40%的时间用于计算,另外60%的时间用于I/O操作。那么增强性能后总的加速比是多少?

解答: 增强比例 = 0.4, 增强加速比 = 10, 总加速比 = $\frac{1}{0.6 + \frac{0.4}{10}} = \frac{1}{0.64} \approx 1.56$

Amdahl定律反映的是收益减少的效果:在只改进一部分的计算性能时,其加速比增量随着改进的增多而减小。Amdahl定律的一个重要推论是,如果某一增强措施仅对任务的一定比例的部分有作用,那么该任务总的加速比不会大于1减去此比例之后求得的倒数。

在应用Amdahl定律时,一个常犯的错误是把“未改进前要改进部分所占的时间比例”和“改进后改进部分所占的时间比例”混淆。如果我们在计算中使用的是改进后改进部分所占的时间比例而不是改进前要改进部分所占的时间比例,结果就会是错误的。

Amdahl定律提供了一种判断增强措施可以提高多少性能的指标,以及如何分配资源才能够改进性价比的途径。很明显,其目标是要使资源花费与时间花费成比例。Amdahl定律对于比较两种设计方案的总体系统性能十分有用,但也可以像下面例子所示,用Amdahl定律比较两种设计方案。

例题 求平方根是图形处理器中经常用到的一种转换,而求浮点数(FP)平方根的不同实现方法在性能上可能有很大差异,特别是这种差异在为图形处理专门设计的处理器中更加明显。假定求浮点数平方根(FPSQR)的操作在某台机器上的一个基准测试程序中占总执行时间的20%。一种方法是增加专门的FPSQR硬件,可以将FPSQR的操作速度提高为原来的10倍;另一种方法是提高所有的FP运算指令的执行速度,FP运算指令在总执行时间中占50%,设计小组认为可以把所有FP指令的执行速度提高为原来的1.6倍,从而提高求浮点数平方根操作的速度。试比较这两种方法。

解答：我们可以通过两种计算方法的加速比来比较其效果：

$$\text{加速比}_{\text{FPSQR}} = \frac{1}{(1-0.2) + \frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

$$\text{加速比}_{\text{FP}} = \frac{1}{(1-0.5) + \frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23$$

提高所有FP运算指令性能的总体效果要好一些，因为该程序中浮点操作所占的比重较大。

Amdahl定律也适用于性能评价之外的其他领域。我们可以重新做一下19页的例题（假设已经改善了电源的可靠性，通过备份电源将可靠性从200 000小时提高到830 000 000小时MTTF，或者说提高了4150倍）。

例题 磁盘子系统的故障率为

$$\begin{aligned} \text{系统故障率} &= 10 \times \frac{1}{1\,000\,000} + \frac{1}{500\,000} + \frac{1}{200\,000} + \frac{1}{200\,000} + \frac{1}{1\,000\,000} \\ &= \frac{10+2+5+5+1}{1\,000\,000 \text{ 小时}} = \frac{23}{1\,000\,000 \text{ 小时}} \end{aligned}$$

因此，可以改进的故障率为5除以23每百万小时，或0.22。

解答：可靠性的改进为

$$\text{电源对的改进} = \frac{1}{(1-0.22) + \frac{0.22}{4150}} = \frac{1}{0.78} = 1.28$$

虽然一个模块的可靠性可以提高到惊人的4150倍，但是从系统的观点来看，由改进措施所带来的好处有限。

在上面的例子中，需要知道新的改进后FP操作所耗费的时间，通常直接测量这些时间是很困难的。在下一节中可以用基于公式的方法进行比较，它把CPU执行时间分成三个独立的分量。如果知道一种方案如何影响这三个分量，就能确定这种方案的总性能效果。我们还可以在设计硬件之前用仿真器来测试这些分量。

处理器性能公式

大多数计算机都基于一个恒定速率的时钟，这些离散时间事件称为跳数、时钟跳数、时钟期、时钟、周期或时钟周期等。计算机设计这者通常用时钟周期持续时间（如1 ns）或时钟的频率（如1 GHz）来描述一个时钟周期的时间。一个程序的CPU时间可以用以下两种方法来描述：

$$\text{CPU 时间} = \text{一个程序的 CPU 时钟周期数} \times \text{时钟周期长度}$$

或者

$$\text{CPU 时间} = \frac{\text{一个程序的 CPU 时钟周期数}}{\text{时钟频率}}$$

除执行程序所需的时钟周期数外,我们还需要计算该程序执行的指令数,即指令路径长度或指令数 (IC, Instruction Count)。如果我们知道了一个程序执行的指令数和执行所需的时钟周期数,就可以计算出执行一条指令所需的平均时钟周期数 (CPI, Clock cycles Per Instruction)。由于它比较容易计算,且这一章里我们将讨论一些简单的处理器,所以我们使用 CPI。有时候设计者们也使用 CPI 的倒数,即每个时钟周期内所执行的指令数 (IPC, Instructions Per Clock)。

CPI 公式为

$$CPI = \frac{\text{一个程序的CPU时钟周期数}}{\text{该程序的指令数}}$$

这一处理器性能指标不论是对不同的指令还是对不同的实现都提供了直观且深入的衡量标准,在接下来的四章中我们会频繁地使用它。

将上式中的指令数转换之后,时钟周期数可定义为 $IC \times CPI$, 这使我们能够在执行时间公式中使用 CPI:

$$CPU \text{ 时间} = \text{指令数} \times CPI \times \text{时钟周期}$$

或者

$$\text{时间} = \frac{IC \times CPI}{\text{时钟频率}}$$

将第一个公式展开成度量单位后,可以看出各项的组合方式:

$$\frac{\text{指令数}}{\text{程序数}} \times \frac{\text{时钟周期数}}{\text{指令数}} \times \frac{\text{秒}}{\text{时钟周期数}} \times \frac{\text{秒}}{\text{程序数}} = CPU \text{ 时间}$$

正像上面的公式显示的那样,处理器性能和三个因素有关:时钟周期、每条指令执行所需的时钟周期数和程序的指令数。此外,这三个因素对 CPU 时间的影响是相同的,这三个因素中的任何一个改进 10%, 时间就会改进 10%。

遗憾的是,孤立地改变一个参数是很困难的,因为改变各因素的技术是相互关联的:

- 时钟周期: 由硬件技术和计算机组成决定。
- CPI: 由计算机组成和指令系统的系统结构决定。
- 指令数: 由指令集系统结构和编译器技术决定。

幸运的是,许多能提高计算机性能的技术主要影响以上三个因素中的一个,而对另外两个因素有较小的影响或是可预测的影响。

对设计处理器来说,有时也用下面的方式计算总的处理器时钟周期:

$$CPU \text{ 时钟周期数} = \sum_{i=1}^n IC_i \times CPI_i$$

其中, IC_i 代表指令 i 在一个程序中的执行次数, CPI_i 代表执行指令 i 所需要的平均时钟周期数。这一形式也可以用来表示 CPU 时间:

$$CPU \text{ 时间} = \left(\sum_{i=1}^n IC_i \times CPI_i \right) \times \text{时钟周期}$$

总的 CPI 也可以表示成

令数,即指令路径数
执行所需的时钟周期
(Instruction)。由于

$$CPI = \frac{\sum_{i=1}^n IC_i \times CPI_i}{\text{总的指令数}} = \sum_{i=1}^n \frac{IC_i}{\text{总的指令数}} \times CPI_i$$

有时设计者们后一个求 CPI 的公式是用每条指令的 CPI 乘以该指令在全部指令中所占的比例(如 $IC \div \text{指令总数}$)。CPI 需要通过测量得到,而不是根据一些手册中查到的值来推算,因为必须要考虑流水线 Cache 缺失以及其他存储器效率问题。

图 1.28 页提到过的例子,现在改用指令的执行频度和指令 CPI 来分析,这在实际中可以通过使用硬件仪器来获得。

且深入的衡量标准假设我们有如下的测量值:

我们能够在执行时
指令(包括 FPSQR)的执行频度 = 25%
指令的平均 CPI = 4.0

其他指令的平均 CPI = 1.33

FPSQR 指令的执行频度 = 2%

FPSQR 指令的 CPI = 20

定有两种备选的设计方案,一种是把 FPSQR 的 CPI 减至 2,另一种是把所有 FP 的 CPI 减至 1.5。接下来我们用处理器性能公式比较这两种方案。

答:首先,我们观察到只有 CPI 发生了变化,时钟频率和指令数保持不变。首先计算没有任何改变的原始 CPI:

$$CPI_{\text{original}} = \sum_{i=1}^n CPI_i \times \left(\frac{IC_i}{\text{总的指令数}} \right) \\ = (4 \times 25\%) + (1.33 \times 75\%) = 2.0$$

指令执行所需的
这三个因素中的任

在原来的 CPI 基础上减去由于增强了 FPSQR 功能而节省的时间,就可以计算出增强 FPSQR 方案的 CPI:

$$CPI_{\text{with new FPSQR}} = CPI_{\text{original}} - 2\% \times (CPI_{\text{old FPSQR}} - CPI_{\text{of new FPSQR only}}) \\ = 2.0 - 2\% \times (20 - 2) = 1.64$$

可以用同样方法计算增强全部 FP 方案的 CPI,或通过把 FP 的 CPI 值和非 FP 的 CPI 值加起来得到。利用后的一种方法的计算如下:

$$CPI = (2.5 \times 25\%) + (1.33 \times 75\%) = 1.62$$

因为增强全部 FP 方案的 CPI 较小,所以这种方案的性能更好。而增强全部 FP 的加速比为

$$\text{Speedup}_{\text{new FP}} = \frac{\text{CPU 时间}_{\text{original}}}{\text{CPU 时间}_{\text{new FP}}} = \frac{IC \times \text{时钟周期长度} \times CPI_{\text{original}}}{IC \times \text{时钟周期长度} \times CPI_{\text{new FP}}} \\ = \frac{CPI_{\text{original}}}{CPI_{\text{new FP}}} = \frac{2.00}{1.625} = 1.23$$

的平均时钟周期数

可以看到这一结果与前面所用 Amdahl 定律计算出的结果是相同的。

在处理器性能公式中,经常可以测量其中的某个分量,这是处理器性能公式比 Amdahl 定律优越的地方。我们一般很难直接测量某种指令的执行时间在总执行时间中所占的比例,而是通过计算指令的执行次数与指令的执行时间的乘积间接得到。如果计算的出发点是指令的执行次数和 CPI,那么计算处理器性能就更加便利。

为了能够用处理器性能公式作为设计工具,我们必须能够测量多种因素。对于一个已有的处理器,得到其执行时间是很容易的,同时时钟频率也是已知的。问题在于计算出指令条数或者 CPI。大部分较新的处理器含有计算已执行代码和时钟频率的计数器。通过周期性地访问这些计数器,并把执行时间和指令条数与代码片断联系在一起,能够帮助程序员更清楚地理解和修改应用程序的性能。通常设计者或程序员希望在比细粒度更为深入的层面上了解性能。例如,他们想知道何为 CPI,在这种情况下就需要使用模拟技术,例如面向处理器的模拟技术。

1.10 综合:性能和性价比

在每一章的最后部分,我们都有一节“综合:……”,用实际的例子来说明这一章的原理。这一节中我们考察性能和性价比的测量,首先用 SPEC 基准测试程序测试桌面计算机系统,然后用 TPC-C 基准测试程序测试服务器。

桌面计算机和机架式系统的性能和性价比

虽然有很多测试程序适合桌面计算机系统,但其中大多数是与操作系统或系统结构相关的。在这一节中,我们将使用 SPEC CPU2000 定点和浮点套件测试一系列桌面计算机系统处理器的性能和性价比。正如图 1.14 提到的, SPEC CPU2000 以 Sun Ultra 5 为基准,使用几何平均数来分析处理器性能,在这种情况下,数值越大表明性能越好。

图 1.15 给出了包括处理器和价格在内的 5 个系统。每个系统都配置了一个处理器、1 GB 的 DDR DRAM (如果支持则带有 EEC 校验)、大约 80 GB 的硬盘和以太网连接。桌面计算机系统还有快速图形卡和监视器,机架式系统则没有。价格上的巨大差异是由以下因素造成的:处理器价格、操作系统差异 (Linux 或 Microsoft OS 和厂商指定 OS 的比较)、系统的可扩展性和商品效应 (1.6 节提到过)。

供应商/型号	处理器	时钟频率	二级 Cache	类型	价格 (美元)
Dell Precision Workstation 380	Intel Pentium 4 Xeon	3.8 GHz	2 MB	桌面	3346
HP ProLiant BL25p	AMD Opteron 252	2.6 GHz	1 MB	机架式	3099
HP ProLiant ML350 G4	Intel Pentium 4 Xeon	3.4 GHz	1 MB	桌面	2907
HP Integrity rx2620-2	Itanium 2	1.6 GHz	3 MB	机架式	5201
Sun Java Workstation W1100z	AMD Opteron 150	2.4 GHz	1 MB	桌面	2145

图 1.15 来自 3 个厂商、使用不同微处理器的 5 种不同的桌面计算机和机架式系统,显示了它们的处理器、时钟频率、二级 Cache 大小和售价。图 1.16 给出了确切的性能和性价比。所有的系统都配置有 1 GB 的 ECC SDRAM 和大约 80 GB 的硬盘。除了一般因素,还有很多导致价格巨大差异的因素。首先,系统提供了不同级别的可扩展性 (Sun Java 工作站的可扩展性最差, Dell 系统适中, HP BL25p 刀片服务器最好)。第二,处理器的成本至少相差 2 倍,其主要原因是二级 Cache 的大小以及更大的晶片导致更高的价格。2005 年,每个 Opteron 的售价在 500~800 美元, Pentium 4 Xeon 的售价在 400~700 美元 (取决于时钟频率和 Cache 大小)。Itanium 2 的晶片大小比其他的要大很多,所以其成本至少是其他的 2 倍。第三,操作系统差异 (Linux 或 Microsoft OS 和厂商指定 OS 的比较) 可能会影响到最后的价格。这些价格来自于 2005 年 8 月份的数据

图 1.16 给出了用 SPEC CINT2000 和 CFP2000 作为度量标准的 5 种系统的性能和性价比。图的右侧是性价比，给出了每 1000 美元的 CINT 或 CFP。注意在各种情况下，相对于基准计算机来说，浮点计算性能总是超过定点计算性能。

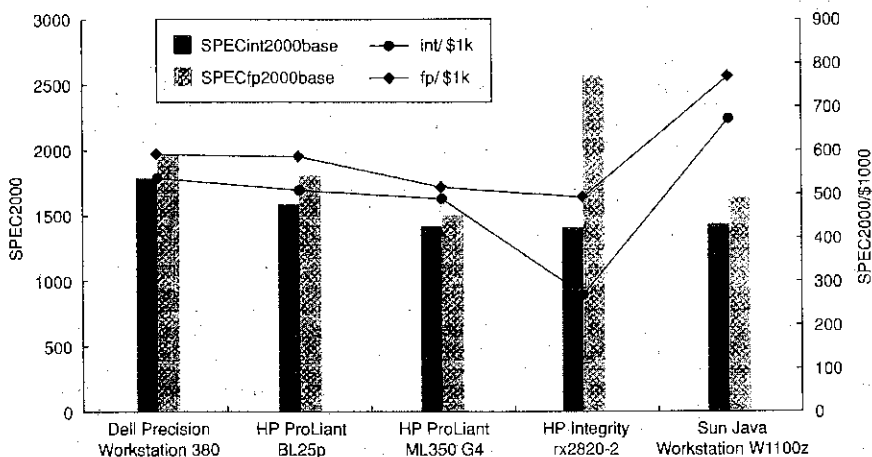


图 1.16 图 1.15 所示的 5 个系统在 SPEC CINT2000 和 CFP2000 基准测试程序下的性能和性价比。性价比是用每 1000 美元系统成本的 CINT2000 和 CFP2000 性能来描述的。这些性能数据是 2006 年 1 月份的数据，价格是 2005 年 8 月份的数据。该评测结果可以在 www.spec.org 上找到

基于 Itanium 2 的设计具有最高的浮点计算性能，但也有最高的成本，结果是拥有最低的每 1000 美元的性能表现（浮点计算为 1.1~1.6，定点计算性能为 1.8~2.5）。基于 3.8 GHz Xeon（2 MB 的二级 Cache）的 Dell 计算机在 CINT 下测得的性能最高，且在 CFP 下测得的性能次高，但比起基于 2.4 GHz AMD Opteron（1 MB 的二级 Cache）的 Sun 产品还是要贵很多，后者在性价比方面是 CINT 和 CFP 两项的“双料冠军”。

事务处理服务器的性能和性价比

最大的服务器市场之一是联机事务处理（OLTP）。OLTP 的标准工业基准测试程序是 TPC-C，它依靠数据库系统来实现查询和更新。5 个因素使得 TPC-C 的性能特别引人注目。首先，TPC-C 是实际应用的一个很合理的近似模拟，尽管这使得测试程序的安装很复杂也很费时，但是它的结果对实际的 OLTP 系统很有指导作用。第二，TPC-C 测试整体系统性能，包括硬件、操作系统、I/O 系统和数据库系统，这使得基准程序能够为实际性能提供更多的预测。第三，运行基准程序和报告执行时间的规则非常完备，所以结果中的数字可比性更强。第四，由于基准程序的重要性，计算机系统销售商对 TPC-C 运行的结果十分看重。第五，销售商需要同时报告性能和价格，以便能够同时考察它们。对于 TPC-C，性能用每分钟的事务数（TPM）来衡量，而性价比用每一美元的 TPM 来衡量。

图 1.17 列出了 10 个性能或性价比排在前面的系统的特性。图 1.18 中用对数刻度给出了精确的性能表示，用线性刻度给出了性价比表示。磁盘的数量是由满足性能目标的每秒钟 I/O 个数所决定的，而不是由需要运行基准测试程序的存储容量所决定的。

性能最高的系统是 IBM 的 64 节点共享存储器多处理器系统，其成本大约为 1700 万美元之多。即使将它一半节点的多处理器系统，价格要贵了一倍，速度也快了一倍；比起处于第三位的 HP 的集群，更是要快 3 倍之多。这五种计算机分别为每台处理器配备了 35~50 个磁盘和 16~20 GB 的 DRAM。第 4 章将讨论多处理器的设计，第 6 章和附录 E 将介绍集群。

供应商/型号	处理器	存储器	存储系统	数据库/操作系统	价格(美元)
IBM eServer p5 595	64 IBM POWER 5 @ 1.9 GHz, 36 MB L3	64 cards, 2048 GB	6548 disks 243 236 GB	IBM DB2 UDB 8.2/ IBM AIX 5L V5.3	16 669 230
IBM eServer p5 595	32 IBM POWER 5 @ 1.9 GHz, 36 MB L3	32 cards, 1024 GB	3298 disks 112 885 GB	Oracle 10g EE/ IBM AIX 5L V5.3	8 428 470
HP Integrity rx5670 Cluster	64 Intel Itanium 2 @ 1.5 GHz, 6 MB L3	768 dimms, 768 GB	2195 disks, 93 184 GB	Oracle 10g EE/ Red Hat E Linux AS 3	6 541 770
HP Integrity Superdome	64 Intel Itanium 2 @ 1.6 GHz, 9 MB L3	512 dimms, 1024 GB	1740 disks, 53 743 GB	MS SQL Server 2005 EE/MS Windows DE 64b	5 820 285
IBM eServer pSeries 690	32 IBM POWER4+ @ 1.9 GHz, 128 MB L3	4 cards, 1024 GB	1995 disks, 74 098 GB	IBM DB2 UDB 8.1/ IBM AIX 5L V5.2	5 571 349
Dell PowerEdge 2800	1 Intel Xeon @ 3.4 GHz, 2 MB L2	2 dimms, 2.5 GB	76 disks, 2585 GB	MS SQL Server 2000 WE/ MS Windows 2003	39 340
Dell PowerEdge 2850	1 Intel Xeon @ 3.4 GHz, 1 MB L2	2 dimms, 2.5 GB	76 disks, 1400 GB	MS SQL Server 2000 SE/ MS Windows 2003	40 170
HP ProLiant ML350	1 Intel Xeon @ 3.1 GHz, 0.5 MB L2	3 dimms, 2.5 GB	34 disks, 696 GB	MS SQL Server 2000 SE/ MS Windows 2003 SE	27 827
HP ProLiant ML350	1 Intel Xeon @ 3.1 GHz, 0.5 MB L2	4 dimms, 4 GB	35 disks, 692 GB	IBM DB2 UDB EE V8.1/ SUSE Linux ES 9	29 990
HP ProLiant ML350	1 Intel Xeon @ 2.8 GHz, 0.5 MB L2	4 dimms, 3.25 GB	35 disks, 692 GB	IBM DB2 UDB EE V8.1/ MS Windows 2003 SE	30 600

图 1.17 在 TPC-C 基准测试程序下, 10 个具有高性能 (表的前半部分, 以每分钟处理的事务数度量) 或高性价比 (表的后半部分, 以每个事务在每分钟的成本度量) 的 OLTP 系统。图 1.18 给出了确切的性能和性价比, 图 1.19 将价格分摊到各个部分上面 (处理器、存储器、存储系统和软件)

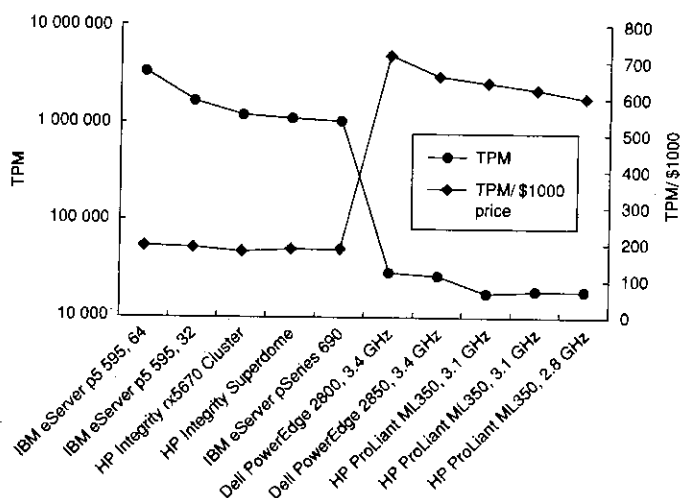


图 1.18 在 TPC-C 基准测试程序下, 图 1.17 中 10 个系统的性能和性价比。性价比用每 1000 美元系统成本的 TPM 表示, 尽管传统的 TPC-C 度量是 $\$/\text{TPM}$ ($715\text{TPM}/\$1000 = \$1.40\$/\text{TPM}$)。这些性能数字和价格来自 2005 年 7 月份的数据。该评测结果可以在 www.tpc.org 上找到

性价比最好的机器是所有基于 Intel Pentium 4 Xeon 处理器的单处理器系统, 尽管所使用的二级 Cache 大小可能不一样。这些系统的性价比比高性能计算机高 3~4 倍。虽然这五台计算机同样为每个处理器平均配置了 35~50 个磁盘, 但每个处理器只使用 2.5~3 GB 的 DRAM。很难讲这是否是

最好的选择, 或者是否仅反映了这些较便宜PC服务器的32位地址空间。因为将存储器加倍只会将价格提高4%, 所以后一种原因是很有可能的。

1.11 谬误和易犯的错误

本书的每一章都将会有一节“谬误和易犯的错误”, 目的是解释清楚读者应当避免的一些常见误解或错误的概念, 我们把这些误解的概念称为谬误。每讨论一个谬误时, 我们都会给出一个反例。我们还会讨论一些易犯的错误。这些错误往往是将一些只在特定环境下才成立的原理一般化。本节以及其他各章中这一节的目的都是帮助读者在设计机器时避免这些错误。

易犯的错误: 忽视 Amdahl 定律。

差不多所有专业计算机设计者都知道 Amdahl 定律, 但我们几乎都犯过类似的错误, 即在测量某个特性的使用之前都要花费巨大的精力去优化它。只有当整体性能的优化微乎其微时, 我们才会记起, 在花费巨大努力优化它之前应该首先测量其实际应用。

易犯的错误: 单点故障。

在前面利用 Amdahl 定律计算了可靠性的改善程度, 其结果表明可靠性取决于系统中最不可靠的部分。无论电源多么可靠, 单个风扇的故障也会降低整个磁盘系统的可靠性。Amdahl 定律的观察结果导致了容错系统经验法则的产生, 即确保系统的每个组成部分都是冗余(备份)的, 这样某一部分的故障就不会导致整个系统的崩溃。

谬误: 处理器的成本在系统成本中占支配地位。

计算机科学是以处理器为中心的, 这可能是因为处理器看起来比存储器或磁盘更加智能化一些, 也可能由于算法一直是以处理器操作数目来衡量的缘故。这让我们误认为处理器的利用是价值的最重要体现。的确, 高性能计算领域常常以所能达到的处理器峰值的百分比来评估算法和系统结构, 这样的做法只有在处理器是主要成本时才有意义。

图 1.19 表明了图 1.17 中各个计算机成本的详细情况, 包括处理器(机箱、电源等)、DRAM 存储器、磁盘存储以及软件。图中甚至将机箱中的片状金属、电源和冷却装置也算在了处理器一栏中, 但结果还是表明处理器的成本并不具有支配地位: 在大规模服务器中, 处理器的成本只占 20% 左右, 在 PC 服务器中更是不到 10%。

谬误: 基准测试程序永远有效。

有几个因素会影响基准测试程序的有效性, 其中某些因素会随着时间而改变。对测试程序有效性影响很大的是测试程序防“破解”的能力, 破解也称为“基准测试程序工程”或“基准程序技巧”。如果一个程序称为标准通用的测试程序, 人们就会不断改进针对它的优化手段以提高评测得分, 此外, 还可以对运行该程序的规则做出各种有利于自己的解释。特别是由几行代码组成的程序内核和小测试程序更容易遭到篡改。

例如, 最初 SPEC89 基准测试程序集中包含了一个称为 matrix300 的小程序内核, 它由 8 个不同的 300×300 矩阵乘法操作组成。在这一程序内核中, 99% 的时间是在执行同一行代码(见 SPEC [1989])。IBM 编译器通过对这一循环代码的优化(利用分块思想, 在第 5 章详细讨论), 使程序的执行速度比使用前一版本的编译器快了 9 倍。这一程序测试的是编译器的性能, 但显然不是测试机

器整体性能的理想手段，因为对一个标准的测试程序来说，不管采用什么样的特殊优化手段，其性能表现都不应该发生太大的差别。

	处理器 + 机箱	存储器	存储 系统	软件
IBM eServer p5 595	28%	16%	51%	6%
IBM eServer p5 595	13%	31%	52%	4%
HP Integrity rx5670 Cluster	11%	22%	35%	33%
HP Integrity Superdome	33%	32%	15%	20%
IBM eServer pSeries 690	21%	24%	48%	7%
高性能计算机的中值	21%	24%	48%	7%
Dell PowerEdge 2800	6%	3%	80%	11%
Dell PowerEdge 2850	7%	3%	76%	14%
HP ProLiant ML350	5%	4%	70%	21%
HP ProLiant ML350	9%	8%	65%	19%
HP ProLiant ML350	8%	6%	65%	21%
高性价比计算机的中值	7%	4%	70%	19%

图 1.19 在图 1.17 中运行 TPC-C 基准测试程序的高端计算机中处理器、存储器、存储系统及软件所占的成本比例。其中存储器只是 DRAM 存储器的成本，所以所有的电源和冷却装置的成本都算在了处理器一项中。TPC-C 包括客户驱动 TPC-C 基准和三年维护（在这里未算）的成本。若算上维护成本，数字还要增加 10%，这和软件维护成本的范围（5%~22%）是不同的。如果算上客户硬件，高性能服务器的价格还要高 2%，PC 服务器还要高 7%

尽管现在的基准测试程序中已将这一程序淘汰，但厂商还是找到了别的窍门来提高系统在测试时的性能表现，例如使用不同的编译器、微处理器和测试程序专用标志等。尽管基准测试程序限制这些手段的使用，但调试版本或优化版本并不限制这些手段。事实上，测试程序专用标志是允许使用的，尽管使用这些标志在通用系统中是不合法的，会产生编译错误。

经过很长的时间后，这些变化甚至可能会使原来精心选择的测试程序过时；Gcc 是唯一从 SPEC89 延用到现在的程序。图 1.13 列出了各个 SPEC 版本的所有 70 个基准测试程序的信息。令人惊讶的是，几乎有 70% 的来自 SPEC2000 或之前版本的程序在新版本中被淘汰了。

谬误：磁盘测定的平均故障时间为 1 200 000 小时或接近 140 年，因此在实际中磁盘永远不可能停止运转。

当前的磁盘厂商常常误导用户。这样的 MTTF 是怎样计算出来的？开始时磁盘厂商在一个房间里放上成千上万个磁盘，并运行几个月，然后计算失效的磁盘数目。他们计算 MTTF 的方法是将所有磁盘总运行时间累加到一起，然后除以失效磁盘的数目。

由此得到的结果远远超过了一个磁盘的寿命期（通常认为是 5 年或 43 800 小时）。现在磁盘厂商说服用户购买他们的磁盘，并平均 5 年更换一次。如果未来的消费者（及其后代）按照这样去做的话，他们在遇上磁盘故障时已经更换了 27 次磁盘，或已经经历了 140 年之久。

一种更有效的测量办法是统计失效磁盘的百分比。假设有 1000 个磁盘以及 1 000 000 小时的 MTTF，每个磁盘每天都运行 24 小时。如果发现失效磁盘就用一个拥有相同可靠性的新磁盘替换，那么一年（8760 小时）下来失效磁盘的数目为

$$\text{失效磁盘个数} = \frac{\text{磁盘数目} \times \text{时间长度}}{\text{MTTF}} = \frac{1000 \text{ 个磁盘} \times 8760 \text{ 小时/驱动器}}{1\,000\,000 \text{ 小时/故障}} = 9$$

即每年 0.9% 的失效率，在磁盘寿命期（5 年）中有 4.4%~5% 的故障率。

设,其性

需要注意的是,这些看起来很高的数字都是假定温度和震动在一定范围内得到的。如果这些因素超出了范围,那么所有由假设得到的结果都不再成立了。最近的一次对于在实际环境中磁盘的调查[Gray and van Ingen; 2005]表明:每年大约有3%~6%的SCSI磁盘发生故障,即大约150 000~300 000小时的MTTF;对于ATA磁盘,这个概率为3%~7%,即大约125 000~300 000小时的MTTF。而宣传的ATA磁盘的MTTF通常是500 000~600 000小时。因此根据这个报告,现实世界中ATA磁盘的MTTF要比厂商提供的MTTF低2~4倍;对于SCSI磁盘,则要低4~8倍。

谬误:峰值性能体现实际性能。

对峰值性能唯一永远正确的定义是,“它是一台计算机绝对不能超越的性能”。图1.20给出了4台程序在4台多处理器系统上达到的峰值性能的比例(5%~58%)。由于峰值与实际性能的差别如此之大,且易受基准测试程序的影响,所以它对预测实际性能意义不大。

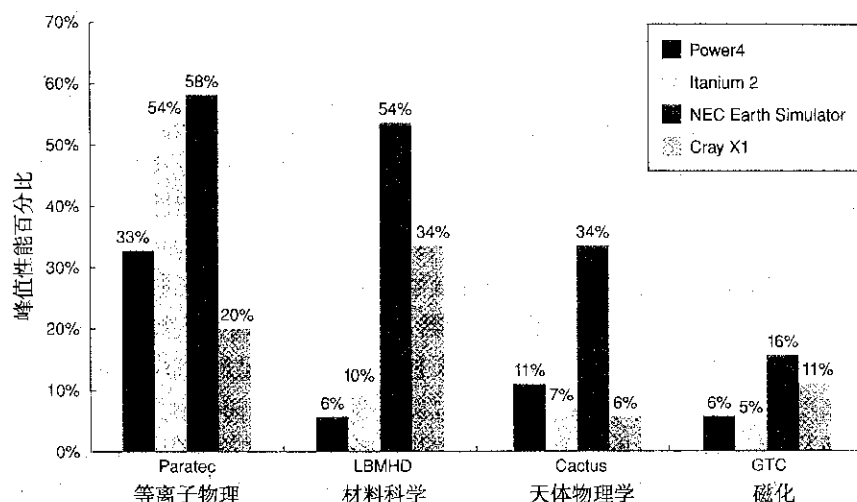


图1.20 4个程序在4台多处理器系统(64个处理器)上所能达到的峰值性能的比例。Earth Simulator和X1是向量处理器(见附录F)。它们不仅可达到较高比例的峰值性能,而且有最高的峰值性能和最低的时钟频率。除了在Paratec程序上偏高之外,Power 4和Itanium 2系统所达到的峰值性能比例都在5%~10%之间。数据来自Oliker等[2004]

易犯的错误:故障检测会降低可用性。

这显然是种误区,因为计算机硬件拥有多种状态,但它们对正确操作不可能总是扮演关键角色。例如,在转移预测器中发生错误不会是灾难性的,只有性能会受到一些损失。

在侧重于指令级并行的处理器中,程序的正确执行并不需要所有操作的参与。Mukherjee等人发现对于在Itanium 2上运行的SPEC2000基准程序来说,只有不到30%的操作处在关键路径上。

在程序中也得到同样的观察结果。如果程序中一个寄存器被标记为“无用”状态,即程序会在它再次被读取之前对它进行写操作,那么这时候出现错误也无关紧要。如果故障检测在“无用”状态的寄存器中检测到暂时的错误,从而导致整个程序的终止,这才是无谓的降低可用性的表现。

2000年Sun公司就曾犯过类似的错误。在它的Sun E3000和Sun E10000系统中,使用了一个奇偶校验但没有错误修正的二级Cache。用来构造Cache的SRAMs有间歇性的故障,这通过奇偶校验可以检测到。如果Cache中的数据未被修改,处理器只是简单地从Cache中再读取一次数据。因为设计者没有使用纠错码(ECC)来保护Cache,在这种情况下操作系统只能报告脏数据错误,并使整个程序终止。但是工程师们发现超过90%的情况下是检查不到任何问题的。

为了降低此类错误的发生频率，Sun 公司修改了 Solaris 操作系统从而实现了“擦净”Cache 的目的，具体方法是通过一个进程将脏数据抢先写入存储器。因为处理器芯片没有足够管脚来支持 ECC 功能，唯一的硬件选择就是在外部 Cache 中复制脏数据，然后使用这个没有奇偶错误的副本来修正错误。

Sun 公司的失误在于能够检测到故障却没有提供修正机制。经过这个教训，Sun 公司肯定不会再出售外部 Cache 中没有 ECC 功能的计算机了。

1.12 结论

本章引入了许多将会在本书中进一步讨论的概念。

在第 2 章、第 3 章中，我们介绍指令级并行（ILP），其中流水线是最简单和最普通的形式。应用 ILP 是设计高速处理器最重要的技术之一。这两章分别介绍用两种不同的方法去开发 ILP，这是一项非常重要且成熟的技术。第 2 章首先讨论在所有章节中用到的一些基本概念，这将有助于对新技术的理解和分析。第 2 章中的例子跨越了大约 35 年的时间，从最初的现代巨型机（IBM360/91）到 2006 年市场上最快的处理器，重点放在动态和运行时应用 ILP 的方法。第 3 章着重介绍第 2 章中提出的 ILP 观点的限制和扩展，包括使用依赖于乱序组织的多线程技术。附录 A 是为初学者提供的流水线的入门材料（我们希望读者能复习这个附录，包括里面的介绍性文章：“计算机组织和设计：硬件/软件接口”）。

第 4 章的重点是如何使用多处理器来实现高性能。这不是单个指令进行重叠，而是使用多处理器来并行执行多个指令流。我们关注的是多处理器系统的主流形式——共享存储器多处理器系统，同时还会介绍其他一些类型。这章将探究的技术主要基于 20 世纪 80 年代到 90 年代提出的一些重要思想。

第 5 章将涉及存储系统设计的各个重要方面。我们将广泛讨论各种在不降低速度的同时增大存储容量的技术。正如在第 2 章到第 4 章所讲的那样，软、硬件结合的技术不仅是设计高性能流水线的关键，而且也是设计高性能存储系统的关键。该章还涉及到虚拟机。附录 C 是为初学者提供的 Cache 入门材料。

在第 6 章里，抛开了以处理器为中心的观点，转而讨论存储系统。这一章采用了类似的量化研究方法，但它基于系统行为的观察，并使用端到端的方法进行分析。该章还讨论了如何利用低成本的磁盘存储技术来提高存取数据的效率。正如我们在前面看到的，这种技术的位成本仅为 DRAM 的 1/100~1/50。该章主要考察磁盘存储系统在典型的以 I/O 操作为主的负载情况下的工作过程，如本章中介绍的 OLTP 测试程序。我们主要探讨基于 RAID 技术的系统，它通过在磁盘阵列中使用冗余技术，来同时满足高速度和高可靠性的要求。最后，该章还将介绍排队论，它是在利用率和时延之间进行折中的主要依据。

随书附带的光盘有更多的材料，这既是为了降低成本，也为了给读者介绍更多的主题。图 1.21 是各个附录的介绍：附录 A、附录 B 和附录 C 包含在本书中，以供读者复习所用；附录 D 是本书内容在嵌入式计算系统中的应用；附录 E 广泛讨论系统互连，包括广域计算机通信网和系统级互连网络，同时也将介绍集群，因为它们在数据库和 Web 服务器应用的稳定性和效率中发挥着越来越大的作用。

附录 F 介绍向量处理器，本书上一版介绍过其中的 NEC Global Climate Simulator，它已经连续几年成为世界上最快的计算机，目前向量处理器的应用正在逐渐增多；附录 G 复习了 VLIW 硬件和软件，但随着 EPIC 的出现（在本书上一版发行不久后），VLIW 正在渐渐淡出人们的视野；附录 I 是唯一从本书第一版保留至今的附录，它涵盖了计算机算术；附录 J 是指令级系统结构的研究，包

Cache的支持。附录K和附录L是习题的参考答案。

附录	章节
A	流水线操作：基本和中级概念
B	指令系统原理和示例
C	存储器层次结构的回顾
D	嵌入式系统（CD）
E	互连网络（CD）
F	向量处理器（CD）
G	VLIW和EPIC的硬件和软件（CD）
H	大规模多处理器和科学应用（CD）
I	计算机算术运算（CD）
J	指令集系统结构综述（CD）
K	历史回顾和参考文献（CD）
L	习题参考答案（在线）

图 1.21 附录列表

1.13 历史回顾和参考文献

随机附送的光盘上的附录K中包括了本书每一章中提到的主要技术的历史回顾。这可以使我们能够通过一系列的机器或有重大意义的项目来回顾技术的发展历程。如果读者致力于研究一种技术或机器的最初发展，或者想进一步深入了解，可以参考附录中的参考文献。对于本章来说，可以参考K.2节——计算机的早期发展，该节主要讨论数字计算机的早期发展以及性能评估方法。

当读者阅读这些历史材料时，会很快意识到和其他工程领域比起来，计算机领域相对“年轻”，它有一个重要优势，就是该领域的众多先驱者目前仍然健在，因此要了解计算机的历史，只需要直接问他们，就这么简单！

1.14 范例分析及习题^①

范例分析 1：芯片制作成本

通过这个范例阐明以下概念：

- 制造成本
- 制造成品率
- 通过冗余实现容错

影响计算机芯片价格的因素很多。新技术实现了性能上的飞跃，芯片也变得越来越小。使用这些技术，设计者能够设计出更小的芯片，或者在一个芯片上设置更多硬件以提供更多的功能。在本例中，将研究包含制造技术、芯片尺寸和冗余在内的设计策略对芯片成本的影响。

1.1 [10/10/讨论]<1.5, 1.5>图 1.22 给出了对目前一些芯片成本影响的相关统计信息。在下面的练习中，要解答关于 AMD Opteron 单芯片处理器和 8 核 Sun Niagara 芯片的问题。

- [10]<1.5>AMD Opteron 的成品率是多少？
- [10]<1.5>8 核 Sun Niagara 处理器的成品率是多少？

^① 本范例分析由 Diana Franklin 提供。

- c. [讨论]<1.4, 1.6>为什么在相同错误率的情况下, Sun Niagara 比 AMD Opteron 的成品率更低?

芯片	晶片面积 (mm ²)	估计错误率 (/cm ²)	制造尺寸 (nm)	晶体管数 (百万)
IBM Power 5	389	0.30	130	276
Sun Niagara	380	0.75	90	279
AMD Opteron	199	0.75	90	233

图 1.22 几种现代处理器的制造成本因素, $\alpha = 4$

- 1.2 [20/20/20/20/20] <1.7>需要决定是否为 IBM Power 5 芯片建造一种新的制造设备。这种设备将花费 10 亿美元。它的好处是可以预期以 2 倍于旧芯片的价格卖出 3 倍于现在的数量。新的芯片面积为 186 mm², 错误率为 0.7/cm²。假设晶片直径为 300 mm, 无论采用什么技术, 一个晶片的成本均为 500 美元。以前销售这种芯片的价格比成本高 40%。
- [20]<1.5>旧的 Power 5 芯片的成本是多少?
 - [20]<1.5>新的 Power 5 芯片的成本是多少?
 - [20]<1.5>每个旧的 Power 5 芯片的利润是多少?
 - [20]<1.5>每个新的 Power 5 芯片的利润是多少?
 - [20]<1.5>如果每月卖出 500 000 个旧的 Power 5 芯片, 需要多长时间可以收回新的制造设备的成本?
- 1.3 [20/20/10/10/20] <1.7>一位在 Sun 公司的同事建议, 既然成品率很差, 销售两套芯片应该会有意义, 一种包含 8 个工作处理器, 另一种包含 6 个。我们将成品率视为在给出错误率的特定区域没有错误发生的概率, 并基于此来完成这个练习。对于 Niagara 来说, 就是分开计算基于各个 Niagara 核的概率 (这可能并不完全精确, 因为成品率公式是基于经验得来的, 而不是关于芯片不同部分错误概率的数学计算)。
- [20]<1.7>使用上面提到的关于错误率的成品率公式, 在一个 8 核芯片中, 单个 Niagara 核 (假设芯片被均匀地划分给各个核) 发生错误的概率是多少?
 - [20]<1.7>一个或两个核发生错误的概率是多少?
 - [10]<1.7>没有核发生错误的概率是多少?
 - [10]<1.7>使用 b 和 c 的答案, 对于每个 8 核的芯片来说, 需要卖出多少 6 核的芯片?
 - [20]<1.7>如果每个 8 核的芯片售价 150 美元, 每个 6 核的芯片售价 100 美元, 每个晶片的成本为 80 美元, 研发预算为 200 万美元, 每个芯片的测试费用为 1.5 美元, 那么需要卖出多少处理器才能收回成本?

范例分析 2: 计算机系统功耗

通过这个范例阐明以下概念:

- Amdahl 定律
- 冗余
- MTTTF
- 功耗

现代系统中的功耗和一系列因素有关, 包括芯片时钟频率、效率、硬盘转速、硬盘所用驱动以及 DRAM。下面的练习中将探究不同的设计思想和 / 或使用场景对电源的影响。

- 1.4 [20/10/20]<1.6>图 1.23 给出了几个计算机系统各个组成部分的功耗情况。在这个练习中, 我们将探究硬盘驱动是怎样影响系统的功耗的。

- a. [20]<1.6>假设每个组件都使用最大的负载，而电源的效率为 70%，那么对于一个拥有 Sun Niagara 8 核芯片、2 GB 184 管脚的 Kingston DRAM 以及两个 7200 rpm 硬盘的服务器来说，其电源必须要提供多少瓦特的功率？
- b. [10]<1.6>如果有一个 7200 rpm 的硬盘，其空闲时间约占总时间的 40%，则其需要消耗多少电源功率？
- c. [20]<1.6>假设每分钟转数是影响磁盘非空闲时间的唯一因素（这是对磁盘性能的过于简单化）。换言之，假设对于同样的请求，一个 5400 rpm 磁盘的处理时间是 10 800 rpm 磁盘处理时间的 2 倍，那么如果执行和 b 相同的事务，5400 rpm 磁盘的空闲时间会占总时间的多少？

部件类型	产品	性能	功耗
处理器	Sun Niagara 8-core	1.2 GHz	峰值为 72~79 W
	Intel Pentium 4	2 GHz	峰值为 48.9~66 W
DRAM	Kingston X64C3AD2 1 GB	184 管脚	3.7 W
	Kingston D2N3 1 GB	240 管脚	2.3 W
硬盘	DiamondMax 16	5400 rpm	读取 / 寻道时为 7.0 W，空闲时为 2.9 W
	DiamondMax Plus 9	7200 rpm	读取 / 寻道时为 7.9 W，空闲时为 4.0 W

图 1.23 几种计算机部件的功耗

- 1.5 [10/10/20]<1.6, 1.7>为服务器群供电的一个关键因素就是散热。如果不能有效地散热，风扇就会将热空气而不是冷气吹回计算机。我们将研究不同的设计思想对散热技术以及系统价格的影响。根据图 1.23 进行计算。
- a. [10]<1.6>一个机架的冷却门要花费 4000 美元，并释放 14 千瓦的热量（清除它们需要另外的费用）。那么使用冷却门你可以为多少拥有 Sun Niagara 8 核芯片、2 GB 240 管脚的 Kingston DRAM 以及来一个 5400 rpm 硬盘的服务器提供散热服务？
- b. [10]<1.6, 1.8>你想为硬盘提供容错机制，RAID 1 使得磁盘数目加倍。那么在使用单散热器的单机架上可以放置多少系统？
- c. [20]<1.8>在单机架上，每个处理器的 MTTF 为 4500 小时，硬盘 MTTF 为 900 万小时，电源 MTTF 为 30 000 小时。那么对于 8 个处理器的机架来说，其 MTTF 是多少？
- 1.6 [10/10/讨论]<1.2, 1.9>图 1.24 在几个基准测试程序下比较了两种服务器的电源功率功耗和性能。这两种服务器是 Sun Fire T2000（使用 Niagara）和 IBM x346（使用 Intel Xeon 处理器）。
- a. [10]<1.9>计算在每个基准测试程序下每种服务器的性能 / 功耗比率。
- b. [10]<1.9>如果用户主要关心功耗，那么会选哪一种服务器？
- c. [讨论]<1.2>对于数据库基准测试程序来说，系统越便宜，每个数据库操作的成本就越低。这是违反常识的：大系统拥有更大的吞吐量，所以人们就会觉得购买大系统的绝对成本会更高，但是每个操作的成本会更低。如果这是正确的，为什么大的企业数据中心总是购买贵的服务器（提示：参看练习 1.4 找原因）？

	Sun Fire T2000	IBM x346
功率 (W)	298	438
SPECjbb (op/s)	63 378	39 985
功率 (W)	330	438
SPECWeb (composite)	14 001	4348

图 1.24 Sun 功耗 / 性能对比，来自 Sun 公司的报告

- 1.7 [10/20/20/20]<1.7, 1.10>公司内部研究表明单核系统对于其所要求的处理能力已经足够用了,但是仍需要考虑使用双核以节能。
- [10]<1.10>假设应用是100%并行的,则需要将频率降低多少才能得到相同的性能?
 - [20]<1.7>假设电压随频率一样直线下降,使用1.5节的公式,那么与单核系统相比,双核系统需要多少动态电源功率?
 - [20]<1.7, 1.10>现在假设电压不会降到初始电压的30%以下。这个电压称为“电压基数”,任何低于“电压基数”的电压都会丢失状态。如果电压降到了“电压基数”水平,那么并行化程度有多少?
 - [20]<1.7, 1.10>使用1.5节的公式,考虑“电压基数”时,在100%并行率的情况下,与单核系统相比,双核系统需要多少动态电源功率?

范例分析3: Web 服务器中可靠性(及故障)的开销

通过这个范例阐明以下概念:

- TPCC
- Web 服务器的可靠性
- MTTF

本节的习题主要针对于没有可靠 Web 服务器情况下的开销。数据分为两部分:一种给出了关于 Gap.com 的不同统计信息,这个网站曾在 2005 年因为维护而关闭了两个星期[AP 2005];另一种是关于 Amazon.com 的信息,这个网站非但没有关闭,反而在高负荷销售的日子中拥有更好的统计记录。本节习题结合这两方面的数据来评估停工时的经济损失。

- 1.8 [10/10/20/20]<1.2, 1.9>在 2005 年 4 月 24 日,由 Gap.com, OldNavy.com, BananaRepublic.com 所管理的三个 Web 服务器因为改进而停工[AP 2005]。在接下来的两周里这几个站点都无法再访问了。使用图 1.25 的统计信息回答以下问题。

公司	时间段	总额	类型
Gap	2004 年第 3 季度	40 亿美元	销售
	2004 年第 4 季度	49 亿美元	销售
	2005 年第 3 季度	39 亿美元	销售
	2005 年第 4 季度	48 亿美元	销售
	2004 年第 3 季度	1.07 亿美元	在线销售
	2005 年第 3 季度	1.06 亿美元	在线销售
Amazon	2005 年第 3 季度	18.6 亿美元	销售
	2005 年第 4 季度	29.8 亿美元	销售
	2005 年第 4 季度	1.08 亿美元	条目销售
	2005 年 12 月 12 日	360 万美元	条目销售

图 1.25 Gap 和 Amazon 的销售统计。数据编辑于 AP [2005], Internet Retailer [2005], Gamasutra [2005], Seattle PI [2005], MSN Money [2005], Gap [2005] 以及 Gap [2006]

- [10]<1.2>在 2005 年第 3 季度, Gap 的收入为 39 亿美元[AP 2005]。其站点在 2005 年 9 月 7 日恢复[Internet Retailer 2005]。假设每天的在线销售额为 140 万美元,其他保持不变,那么 Gap 在 2005 年第 3 季度的收入是多少?
- [10]<1.2>如果这个停工时期发生在第 4 季度,那么预计停工时间的开销是多少?

c. [20]<1.2>站点恢复时,可以访问的用户数量是受限的。假设只有50%的消费者可以访问网站,而且购买和装配每台服务器的开销为7500美元,那么销售中损失的资金相当于他们每天需要购买及装配多少服务器?

d. [20]<1.2, 1.9>在2004年7月, Gap.com每天有260万访问者[AP 2005],一个用户平均每天要浏览 Gap.com的8.4个网页。假设 Gap.com的高端服务器运行的是 SQL Server 软件,在TPCC基准下每次操作的估计开销为5.38美元,那么 Gap.com每天为支持其在线交易的开销是多少?

1.9 [10/10]<1.8>可靠性的主要测量方法是MTTF。我们现在要探究不同的系统和设计思想对可靠性产生的影响。可以从图 1.25 中得到公司的统计信息。

a. [10]<1.8>有一个 100FIT 的单处理器,则系统的 MTTF 是多少?

b. [10]<1.8>如果系统重启需要一天的时间,则系统的可用性是多少?

1.10 [20]<1.8>假设政府削减预算,计划建造一台超级计算机,它是由习题 1.9 中所提到的便宜处理器组成的系统,而不使用在可靠性上有特殊要求的系统,那么拥有 1000 个处理器的系统,其 MTTF 是多少?假设一个处理器失效,整个系统就发生故障。

1.11 [20/20]<1.2, 1.8>在诸如 Amazon 或者 Gap 的服务器集群中,单一故障不会引起整个系统的崩溃。相反地,它会减少可提供满意服务的请求数量。

a. [20]<1.8>如果一家公司拥有 10 000 台计算机,且当三分之一的计算机发生故障时,整个系统才会发生灾难性的故障,则系统的 MTTF 是多少?

b. [20]<1.2, 1.8>如果想将 MTTF 提高一倍,那么每台机器需要 1000 美元的额外开销,这是否是一个好的商业选择?证明你的观点。

范例分析 4: 性能

通过这个范例阐明以下概念:

- 算数平均值
- 几何平均值
- 并行
- Amdahl 定律
- 加权平均

在本节的练习中需要弄清楚图 1.26 的含义,它给出了所选的处理器和一个虚构处理器(处理器 X)的性能数据(源自 www.tomshardware.com)。每个系统都运行了两套基准测试程序。一套基准测试程序用于测试存储器层次结构,并得到系统存储器的速度信息;另外一套 Dhrystone 基准测试程序不涉及存储器层次结构,而是一套面向 CPU 的基准测试程序。使用两套基准测试程序的目的是能够更清晰地说明不同的设计思想对存储器和 CPU 性能的影响。

1.12 [10/10/讨论/10/20/讨论]<1.7>使用图中的原始数据进行计算,以便弄清楚不同的测试方法是怎样影响最后结果的(做这些习题时最好使用电子表格软件)。

a. [10]<1.8>创建一个类似于图 1.26 的表格,但对于每个基准测试程序,要把结果基于 Pentium D 进行规格化。

b. [10]<1.9>计算每个处理器性能的算数平均数。使用原始性能和在 a 中规格化后得到的性能。

- c. [讨论]<1.9>使用 b 中得到的结果,能得到不同处理器之间相关性能的冲突情况吗?
- d. [10]<1.9>使用 Dhrystone 基准测试程序,计算双核处理器以及单核处理器规格化性能的几何平均数。
- e. [20]<1.9>画一幅二维图, x 轴表示 Dhrystone 基准测试程序, y 轴表示存储器基准测试程序。
- f. [讨论]<1.9>在 e 所示图中,双核处理器的性能增加处于什么区域?根据并行处理和系统结构的知识,解释所得到的结果。

芯片	核的数量	时钟频率 (MHz)	存储器 性能	Dhrystone 性能
Athlon 64 X2 4800+	2	2400	3423	20 718
Pentium EE 840	2	2200	3228	18 893
Pentium D 820	2	3000	3000	15 220
Athlon 64 X2 3800+	2	3200	2941	17,129
Pentium 4	1	2800	2731	7621
Athlon 64 3000+	1	1800	2953	7628
Pentium 4 570	1	2800	3501	11 210
Processor X	1	3000	7000	5000

图 1.26 两套基准测试程序下几个处理器的性能

- 1.13 [10/10/20]<1.9>假设你的公司要在单核处理器和双核处理器之间作出选择。图 1.26 给出了两套基准测试程序下的性能数据。而且已知应用在以存储器为中心的计算机中运行将耗费 40% 的时间,而对于以处理器为中心的计算机,这个比例为 60%。
- a. [10]<1.9>计算基准测试程序的加权执行时间。
- b. [10]<1.9>如果在执行一套面向 CPU 的应用时,不使用 Pentium 4 570 而转为使用 Athlon 64 X2 4800+,则可以期望会得到多少的加速?
- c. [20]<1.9>存储器和处理器计算比率是多少时, Pentium 4 570 和 Pentium D 820 的性能相同?
- 1.14 [10/10/20/20]<1.10>公司刚刚购置了一个新的双核 Pentium 处理器系统,需要面向这个处理器优化所用的软件。要在这个处理器上运行两种应用,但是资源要求是不相等的。第一个应用需要 80% 的资源,第二个只需要 20% 的资源。
- a. [10]<1.10>如果第一个应用的并行率为 40%,计算单独运行该应用时的加速比。
- b. [10]<1.10>如果第二个应用的并行率为 99%,计算单独运行该应用时的加速比。
- c. [20]<1.10>如果第一个应用的并行率为 40%,计算并行运行时的系统总加速比。
- d. [20]<1.10>如果第二个应用的并行率为 99%,计算并行运行时的系统总加速比。

第2章 指令级并行及其开发

“谁第一?”

“美国。”

“谁第二?”

“先生，没有第二。”

“美洲杯”帆船赛每隔几年就要举行一次，这是帆船赛赛后发生在两位观众之间的对话。这为 John Cocke 将 IBM 公司研制的一款处理器命名为“美国”提供了灵感。这个处理器是第一个采用超标量结构的微处理器，是 RS/6000 系列微处理器的前身。

2.1 指令级并行：概念与挑战

26 给出了
行将耗费

自 1985 年以来，所有的处理器都采用流水线方式使指令的执行可以重叠进行以提高性能。由于可以将指令间的关系看做是并行的，因此将指令间的这种潜在重叠称为指令级并行 (ILP, instruction-level parallelism)。在这一章和附录 G 中，我们将讨论一系列技术，这些技术提高了指令序列的并行度，扩展了流水线的基本概念。

与附录 A 中有关流水线的基础内容相比，本章的内容更加深入。如果读者对附录 A 的内容还不熟悉，建议在进入本章之前先对其中的内容进行回顾。

用 Athlon

20 的性能

在本章的开始，我们将首先讨论数据冒险 (Hazard) 和控制冒险带来的限制，进而转到主题，讨论怎样增强编译器和处理器对并行的开发能力。这几节介绍的许多概念将成为第 2 章和第 3 章的基础。实际上，即使读者没有彻底掌握前两节的全部概念，仍然可以理解本章中的一些基础内容，但是这些概念对于本章的后面几节和第 3 章来说都是十分重要的。

这个处理

1. 第一个

开发指令级并行的方法大致可以分为两类：一种方法依赖于硬件，动态地发现和开发指令级并行；另一种方法依赖于软件技术，在编译阶段静态地发现并行。使用基于硬件的动态方法的处理器包括 Intel 的 Pentium 系列，在市场上占据主导地位；而采用静态方法的处理器包括 Intel 的 Itanium，其适用范围局限于科学领域或特定应用环境。

过去几年中，上述两种方法往往会在设计过程中互相交叉。本章将介绍基本概念和这两种方法。第 3 章将着重讨论与指令级并行的限制有关的问题。

在这一节中，将讨论程序和处理器对指令序列并行度的限制，以及程序结构和硬件结构间的映射。要判断程序的特性是否会对性能产生影响，以及在什么情况下会对性能产生影响，程序结构和硬件结构间的映射是关键问题。

流水线机器的 CPI (执行每条指令所用的时钟周期数) 等于基本 CPI 与各种停顿所使用的周期数的和。

即未访问

未访问

流水线 CPI = 理想流水线 CPI + 结构停顿 + 数据冒险停顿 + 控制停顿

理想流水线 CPI 是指在执行过程中可达到的最大性能。通过减小等式右边各项的值，可以将总的流水线 CPI 减到最小，或者换种说法，即提高 IPC (每时钟周期指令数量)。参照上述等式，可以以减

小总体CPI的哪一部分为依据来描述各种技术的特征。我们将在这一章和附录G以及附录A所涵盖的主题中讨论这些技术,如图2.1所示。在本章中将会看到,这些用来减小理想流水线CPI的技术将使处理冒险的问题变得更加重要。

技术	减少理想CPI中的哪部分	章节
直传和旁路	潜在的数据冒险停顿	A.2
延迟转移和简单转移调度	控制冒险停顿	A.2
基本动态调度(记分板)	真相关引起的数据冒险停顿	A.7
重命名动态调度	数据冒险停顿、反相关和输出相关引起的停顿	2.4
转移预测	控制停顿	2.3
每个周期发射多条指令	理想CPI	2.7, 2.8
硬件推测	数据冒险和控制冒险停顿	2.6
动态存储器消除二义	涉及存储器的数据冒险停顿	2.4, 2.6
循环展开	控制冒险停顿	2.2
基本编译器流水线调度	数据冒险停顿	A.2, 2.2
编译器相关性分析,	理想CPI, 数据冒险停顿	G.2, G.3
软件流水线, 踪迹调度		
硬件支持编译器推测	理想CPI, 数据冒险停顿, 转换冒险停顿	G.4, G.5

图2.1 在附录A、第2章或附录G中将要讨论的主要技术及它们对总体CPI各部分的影响

什么是指令级并行

本章中将要讨论的所有技术都是用来提高指令序列的并行度的。一个基本块(除入口和出口外没有其他转移的线性指令序列)可达到的并行度是十分有限的。对于典型的MIPS程序来说,动态转移的发生频率平均在15%~25%之间,这意味着每隔3~6条指令就要执行一条转移指令。由于这些指令很可能是相互关联的,因此在一个基本块中可以得到的重叠数有可能会小于基本块的平均大小。为了更显著地提高性能,必须在多个基本块之间开发指令级并行。

为了提高指令级并行,最简单也最常用的方法是将一个循环中的各次迭代并行执行。通常将这类并行称为循环级并行。下面是一个简单的例子,这段程序完成由1000个元素组成的两个数组的相加,它完全可以并行执行:

```
for (i=1; i<=1000; i=i+1)
    x[i] = x[i] + y[i];
```

在这个循环中,每个迭代都可以与其他任何一次迭代重叠执行,尽管在每次迭代内部几乎没有重叠的可能。

在这一章中,我们将讨论一些能够将这类循环级并行转化为指令级并行的技术。这些技术基本上都是通过将循环展开进行实施的,或者通过编译器以静态的方式展开(见下一节),或者由硬件动态地展开(见2.5节和2.6节)。

开发循环级并行的另一种重要方法是使用向量指令(见附录F)。向量指令通过对数据的并行操作实现数据级并行。例如,上面这段代码可以在一些向量处理器上用4条指令执行:两条指令用来从存储器中读取向量x和向量y,一条指令用来将两个向量相加,一条指令用来将结果写回。这些指令当然可以采用流水线的方式执行,虽然存在相对较长的时延,但这些时延是可以重叠的。

虽然向量思想的出现要早于许多指令级并行技术,但在通用处理器市场采用指令级并行技术,处理器则几乎完全取代了基于向量的处理器。而向量指令系统则在图形、数字信号处理和多媒体应用领域的就业前景被一致看好。

数据相关和冒险

要判断一段程序蕴涵多少并行度,以及这种并行度可被开发到什么程度,判定指令间的相关性是一个关键问题。特别是在开发指令级并行时必须明确哪些指令是可以并行执行的。两条指令是并行的,是指在流水线有充足资源的情况下(因此不存在结构冒险),这两条指令可以在任意深度的流水线上并行执行而不会产生停顿。两条指令是相关的,是指它们无法并行,并且只能以顺序的方式执行,尽管在它们之间可能存在部分重叠。指令之间是否存在相关性是决定指令能否并行执行的关键因素。

数据相关

相关有三种不同的类型:数据相关(也称为真数据相关)、名字相关和控制相关。如果下面的条件之一成立,则指令 j 数据相关于指令 i :

- 指令 j 可能会引用指令 i 的结果。
- 指令 j 数据相关于指令 k ,而指令 k 数据相关于指令 i 。

其中第二个条件表明:如果两条指令之间存在由第一种类型的相关组成的相关链,则这两条指令也是相关的。这条相关链甚至可以贯穿整个程序。请注意,在单独一条指令存储器在的相关(如 ADD R1, R1, R1)不是真正意义上的相关。

例如下面的这段MIPS代码序列,它对存储器中的数组元素做递增操作,0(R1)存放数组元素的首地址,8(R2)存放数组元素的尾地址,F2中存放标量S(为简单起见,在本章的例题中,我们忽略了延迟转移的影响)。

```

Loop:   L.D    F0,0(R1)    ; F0 = 数组元素
        ADD.D  F4,F0,F2    ; 加上 F2 中的标量
        S.D    F4,0(R1)    ; 保存结果
        DADDUI R1,R1,-8    ; 指针减 8 个字节
        BNE    R1,R2,LOOP  ; R1 ≠ R2 时转移
  
```

上述代码中存在的数据相关涉及浮点数据:

```

Loop:   L.D    F0,0(R1)    ; F0 = 数组元素
        ADD.D  F4,F0,F2    ; 加上 F2 中的标量
        S.D    F4,0(R1)    ; 保存结果
  
```

和定点数据:

```

DADDUI R1,R1,-8    ; decrement pointer
        ↓
        ; 指针减 8 个字节
        BNE    R1,R2,Loop ; R1 ≠ R2 时转移
  
```

上述两段指令是数据相关的,正如箭头所标注的那样,这两段代码中的每条指令均与前面的一条指令相关。此处和以后的例子中出现的箭头均表示为了确保正确执行而必须保持的指令顺序。箭头尾部的指令必须在箭头指向的指令之前执行。

两条数据相关的指令是不能同时执行或者完全重叠的。相关意味着在两条指令之间可能存在一条由一个或多个数据冒险组成的相关链(见附录A中对数据冒险的简要描述,我们将在后面几页进行准确定义)。同时执行数据相关的指令会使内部互锁流水线的处理器检测到冒险,造成停顿,从而减小甚至消除指令间的重叠度。在一个没有内部互锁的、依赖编译器动态调度的处理器中,编译器不能以完全重叠的方式调度存在相关性的指令,因为这样做会导致程序的错误执行。指令序列的数据相关是生成该指令序列的源代码中存在数据相关的反映。这种原始的数据相关性应被保留。

data
Name (anti)
Name (output)
Control

是否存在相关取决于程序的性质。而一个给定的相关是否会引起冒险以及冒险是否会造成停顿则取决于流水线的结构。要确定指令级并行开发的程度,了解这一点是个关键问题。

数据相关传递了三个方面的信息:(1)数据相关表明存在冒险的可能,(2)数据相关决定了必须遵循的执行顺序,(3)数据相关决定了可以达到并行度的上限。我们将在第3章中讨论相关的问题。

由于数据相关会限制指令级并行的可开发度,所以在本章中我们将着重讨论克服这些限制的方法。克服相关性的方法主要有两种:在保持相关的情况下避免冒险;通过转换代码的方法消除相关。调度代码是在不改变相关性的条件下避免冒险的首要方法,这种调度既可以由编译器来完成,也可以由硬件来完成。

数据可以通过寄存器,也可以通过存储单元在指令间流动。尽管涉及到转移和程序正确性的问题时,编译器和硬件的功能会受到一定的限制,情况也要复杂一些,但是总的说来,由于在指令中寄存器的名字是固定的,因此当数据流发生在寄存器中时,检测数据相关的工作要相对简单明了。

相比之下,当数据流发生在存储单元中时,检测相关的工作就要困难许多。这是由于表面上看起来完全不同的两个地址有可能指向同一个存储器单元。比如100(R4)和20(R6)就有可能表示相同的地址。同时,load和store的有效地址会随着指令的每次执行而变化(因此20(R4)和20(R6)有可能是不同的),这使得检测相关的工作变得更加复杂。

在本章中,我们将讨论涉及到存储单元的数据相关检测,但是将会看到,这些技术并不是完美的,它们也存在各自的缺陷。对于循环级并行来说,通过编译器检测相关性是其关键技术,我们将在附录G中讨论这个问题。

名字相关

相关的第二种类型是名字相关。名字相关发生在使用相同的寄存器或存储单元(称为名字)的两条指令之间,但是在名字相关的指令间不存在数据流。名字相关有两种类型(假设在程序顺序中,指令*i*位于指令*j*之前):

1. **指令*i*和*j*之间的反相关:**指令*i*读一个寄存器或存储器单元,而指令*j*写该寄存器或存储器单元。在这种情况下,为了保证指令*i*读到正确的值,必须保护原始的指令执行顺序。在上页的例子中,S.D和DADDIU之间在寄存器R1上存在反相关。
2. **输出相关:**当指令*i*和指令*j*写相同的寄存器和存储器单元时,为了保证该寄存器或存储器单元的值最后是由指令*j*写入的,必须保护指令的执行顺序。

反相关和输出相关都是名字相关,与数据相关相反,名字相关的指令之间不存在数据流。名字相关不是真正的相关,因此我们可以通过改变指令中使用的名字(寄存器和存储单元)来化解指令间的矛盾,从而使名字相关的指令可以并行执行或改变顺序。

当操作数保存在寄存器中时,这种重命名方法更容易实施,我们称之为寄存器重命名。寄存器重命名可以由编译器静态实施,也可以由硬件动态实施。在讨论由转移引起的相关性之前,我们先来研究一下相关性和流水线的数据冒险之间的关系。

数据冒险

如果两条相关指令在执行顺序中足够接近,使得它们在执行期间产生重叠,并且这种重叠会使访问相关操作数的顺序发生改变,这时就会发生冒险。由于存在相关,我们必须保护程序顺序,即由源程序确定的指令执行序列。不论是软件技术还是硬件技术,它们的目标都是在不改变程序输出的情况下尽力开发并行度,当程序顺序的改变会影响程序输出时,它们必须保护程序顺序。检测和避免冒险可以确保必须保持的程序顺序不被破坏。

依据指令的读写访问顺序,数据冒险(在附录A中给出了非正式的描述)可以被分为三类。按惯例,一般以流水线必须保护的程序顺序来对冒险命名。比如假设有指令*i*和指令*j*,其中指令*i*在程序顺序中位于指令*j*之前,可能的数据冒险包括:

- **RAW (写后读)**: *j* 试图在 *i* 写一个数据之前读取它。这时 *j* 将错误地读出旧值。RAW 是最常见的冒险类型,它对应于真数据相关。在这种情况下,为了保证指令 *j* 读取指令 *i* 写入的值,必须保持程序顺序。
- **WAW (写后写)**: *j* 试图在 *i* 写一个数据之前写该数据。这时,如果执行顺序错误,那么当写操作结束时,留下的值将是 *i* 写的结果,而程序的本意是在这里留下 *j* 写的值。这类冒险对应于输出相关。WAW 只在特定类型的流水线中才会发生,这些流水线允许在多个流水段进行写操作,或者允许后续指令在前面的指令停顿的情况下继续执行。
- **WAR (读后写)**: *j* 试图在 *i* 读一个数据之前写该数据。这时, *i* 将错误地读出新值。这类冒险是由反相关引起的。WAR 不会发生在静态流水线中,即使对深度流水线和浮点流水线来说也是这样。这是由于在静态流水线中,所有的读操作(在 ID 流水段)发生得早,写操作(在 WB 流水段)发生得晚(见附录 A)。只有在指令流水的过程中一些指令的写操作提前完成、或者其他指令的读操作滞后完成、或者指令顺序被改变的情况下,才会发生 WAR 冒险,我们还会在这一章中继续讨论这个问题。

请注意 RAR (读后读) 不产生冒险。

控制相关

最后一种类型的相关是控制相关。控制相关决定了与转移指令有关的指令的执行顺序,从而使与转移有关的指令只在应当被执行时按程序顺序执行。除第一个基本块外,程序中的每条指令都与某些转移指令控制相关,一般来说,必须保护这种控制相关性以确保正确的程序顺序。控制相关最简单的例子是在转移的 if 语句的 then 部分中的语句相关。比如下面的代码片断:

```
if p1 {
    S1;
};
if p2 {
    S2;
};
```

S1 与 p1 控制相关, S2 与 p2 而不是 p1 控制相关。

一般来说,控制相关会带来两类限制:

1. 与某一转移相关的指令不能被移动到该转移之前,这样的移动会使指令的执行不再受控于该转移。例如,我们不能将 if 语句的 then 部分中的指令移动到 if 语句之前。
2. 与某一转移无关的指令不能被移动到该转移之后,这样的移动会使指令的执行受到该转移的控制。例如,我们不能将 if 语句之前的指令移动到受控于该 if 语句的 then 部分中。

处理器严格保护程序顺序的同时,当然也确保了控制相关性不被破坏。但有时候,违背控制相关性,执行一些不该被执行的指令,并不会影响程序的正确性。其实控制相关并不是必须被保护的关键因素。实际上,为了确保程序的正确性,异常行为和**数据流**——通常通过维护数据相关和控制相关来对它们进行保护——这才是最关键的因素。

保护异常行为意味着对指令执行顺序的任何改变都不应改变程序中的异常的发生。通常,它被放宽到指令执行顺序的改变不应在程序中引起新的异常。下面的代码序列说明了如何通过维护数据相关和控制相关来保护异常行为:

```

DADDU    R2,R3,R4
BEQZ     R2,L1
LW       R1,0(R2)

L1:

```

在这个例子中,如果不保护 R2 上的数据相关性,程序的结果就会被改变,这是显而易见的,但不那么容易看出。通过分析发现,如果我们忽略控制相关性而将 load 指令移动到转移之前,则可能会引起存储器保护异常。注意,这里是控制相关带来的限制妨碍我们交换 BEQZ 指令 and LW 指令,而与数据相关性无关。如果需要改变指令的顺序(仍然保持数据相关性),我们可以忽略转移引起的异常。在 2.6 节中,我们将讨论一种硬件技术——推测,这种技术可以使我们克服这类异常问题。附录 G 中讨论了通过软件技术支持推测的问题。

通过维护数据相关和控制相关而保护的第二个因素是数据流。数据流是指数据值在产生结果的指令和使用结果的指令之间的实际流动。转移使数据流是动态的,这是由于转移使得一条给定指令的数据源可能来自不同的地方。这时仅仅维护数据相关性是不够的,因为这条指令可能与之前的多条指令数据相关,由哪一条指令传送数据值是由程序顺序决定的,而要保证程序顺序就必须维持控制相关。

以下面的代码片段为例:

```

DADDU    R1,R2,R3
BEQZ     R4,L
DSUBU    R1,R5,R6

L:       ...

OR       R7,R1,R8

```

在这个例子中,OR 指令中使用的 R1 的值取决于转移指令是否被选中。这里只对数据相关性进行维护是不够的。OR 指令数据相关于 DADDU 指令和 DSUBU 指令,但是只靠保护执行顺序并不足以确保程序的正确执行。

在指令的执行过程中,必须保证数据流不被破坏:在转移指令未被选中的情况下,OR 指令引用的是 DSUBU 指令对 R1 的计算结果;而如果转移指令被选中,则应当引用 DADDU 指令对 R1 的计算结果。我们通过维持 OR 指令与转移指令间的控制相关避免了对数据流的非法改动。鉴于相同的原因,DSUBU 指令也同样不能被移动到转移指令之前。推测技术可以帮助我们处理异常问题,同时在维持数据流不变的情况下降低控制相关的影响,这些问题将在 2.6 节中进行讨论。

有些情况下改变控制相关不会影响异常行为和数据流,如下面的代码段:

```

DADDU    R1,R2,R3
BEQZ     R12,skip
DSUBU    R4,R5,R6
DADDU    R5,R4,R9

skip:    OR      R7,R8,R9

```

如果 DSUBU 指令的目标寄存器(R4)在标号 skip 的指令之后不会被用到(称一个值是否会被后续指令用到的属性为活性),那么,我们就可以在不影响数据流的情况下在转移之前改变 R4 的值,这是由于 skip 后面的代码与 R4 无关,也就是说 R4 在 skip 后失去了活性。在这种情况下,如果 DSUBU 指令不会产生异常,我们就可以将 DSUBU 指令移动到转移之前,因为这样的移动不会影响数据流。

选中转移指令会使DSUBU指令的执行失去意义,但这并不会影响程序的结果。这种类型的代码调度也是推测的一种形式,由于需要编译器推测转移的输出,因此通常也称之为软件推测;在这个例子中,编译器实际上做出了转移不被选中的推测。附录G中会详细讨论编译器的推测机制。当使用“推测”这个词时,是软件推测还是硬件推测一般来说是很明确的;但是当含义不清时,最好还是明确地使用“软件推测”或“硬件推测”进行界定。

要对导致控制停顿的控制冒险进行检测,以保证控制相关不被破坏。有多种硬件技术和软件技术可以减小或消除控制停顿,我们将在2.3节中讨论这些技术。

2.2 支持指令级并行的基本编译技术

这一节将首先讨论一些简单的编译技术,这些技术能够提升处理器对并行度的开发能力。对于使用静态发射或静态调度的处理器来说,这些技术是至关重要的。在这些技术的支持下,我们将简短地讨论静态发射处理器的设计和性能问题。附录G中将介绍一些能够使处理器获得更高指令级并行度的高级编译技术和相关的硬件机制。

基本流水线调度和循环展开

要保持一条流水线不停顿,就要去发现可以流水重叠的不相关的指令序列,并加以充分利用。为了避免流水线的停顿,就要事先找出指令代码中的相关指令并将它们分离,使其相隔的时钟周期能正好等于原来指令在流水执行时的时延。编译器进行这类调度的能力既依赖于程序的指令级并行度,也依赖于流水线中功能单元的时延。在本章中,假设浮点单元的时延如图2.2所示,除非明确地指出有不同的时延。假设流水线采用标准的五段定点流水结构,转移的延迟为一个时钟周期。假设功能单元可以充分流水或重复设置(即可以与流水线深度相同),因此所有类型的操作都可以在任意一个时钟周期内发射而不会造成结构冒险。

产生结果的指令类型	使用结果的指令类型	时延的时钟周期
浮点 ALU 操作	另一个浮点 ALU 操作	3
浮点 ALU 操作	双精度 Store	2
双精度 Load	浮点 ALU 操作	1
双精度 Load	双精度 Store	0

图2.2 本章中使用的浮点操作的时延。最后一列是为避免暂停而必需的间隔时钟周期数。这些周期数与我们将要在浮点单元中见到的平均时延相似。浮点 load 对 store 的时延是0,因为 load 的结果可以旁路给 store 而不会使其暂停。同时,假设一个定点 load 的时延为1,而一个定点 ALU 操作的时延为0

在这一小节中,我们会看到编译器是如何通过转换循环来提高可用的指令级并行度的。以下这个例子除阐述这一重要技术外,还会涉及到在附录G中介绍的更强大的程序转换技术。让我们来看看下面的代码片断,它实现一个数组与标量相加的操作:

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

我们注意到这个循环中的每次迭代都是独立的,因此这个循环是可以并行的。附录G中给出了这个概念的规范化描述,并详细讨论了怎样在编译阶段检测循环迭代之间是否互不相关。首先,来考察这个循环的性能,即在一个如上文所述的MIPS流水线的时延情况下,怎样通过利用并行性来提高性能。

首先要把上述代码转换为 MIPS 汇编语言。在下面的代码片断中, R1 初始时是数组元素的最高地址, F2 中存放标量值 S。寄存器 R2 的值被预先计算出来以保证 8(R2) 是最后一个操作的元素的地址。

下面是上述代码经过直接转换后的 MIPS 汇编代码, 没有为流水线进行调度:

```

Loop:  L.D      F0,0(R1)      ; F0 = 数组元素
        ADD.D   F4,F0,F2     ; 加 F2 中的标量
        S.D     F4,0(R1)     ; 存储结果
        DADDUI  R1,R1,#-8    ; 递减指针
                                ; 8 个字节 (一个双字)
        BNE     R1,R2,Loop   ; R1 ≠ R2 时转移

```

下面是这个循环在如图 2.2 所示的 MIPS 流水线中调度执行的情况。

例题 分别写出在调度之前和调度之后, 这个循环在 MIPS 上的运行情况, 考虑所有的停顿和空闲时钟周期。调度时考虑浮点操作的延迟, 忽略延迟转移。

解答: 在调度之前, 该循环将按下面的方式执行, 消耗 9 个周期:

		发射的时钟周期
Loop:	L.D F0,0(R1)	1
	stall	2
	ADD.D F4,F0,F2	3
	stall	4
	stall	5
	S.D F4,0(R1)	6
	DADDUI R1,R1,#-8	7
	stall	8
	BNE R1,R2,Loop	9

在调度之后, 我们可以将停顿控制在两个时钟周期, 将消耗的时钟周期减小到 7 个:

```

Loop:  L.D      F0,0(R1)
        DADDUI  R1,R1,#-8
        ADD.D   F4,F0,F2
        stall
        stall
        S.D     F4,8(R1)
        BNE     R1,R2,Loop

```

ADD.D 之后的停顿是由 S.D 引起的。

在上面这个例子中, 我们每执行一次循环迭代并将存储结果写回数组元素需要花费 7 个时钟周期, 而其中对数组元素的实际操作只占 3 个周期 (load, add, store)。其余的 4 个时钟周期包括循环的开销 (DADDUI 和 BNE) 和 2 个停顿。为了消除这 4 个时钟周期的影响, 我们需要在循环体中增加有效操作对开销指令的比重。

为了增加有效操作对转移和开销指令的比重, 循环展开是一个简单的方法。展开可以通过多次复制循环体和调整循环中止代码来实现。

循环展开也可以用来改进调度的效果。由于它消除了转移, 因此来自不同迭代的指令可以被一起调度。在这个例子中, 我们可以通过在迭代内加入不相关的指令来消除数据相关引起的停顿。如果在循环展开的过程中只是简单地复制指令, 那么对相同寄存器的使用将导致循环不能被有效地调度。因此, 需要为每次迭代分配不同的寄存器, 而这增加了所需的寄存器数目。

元素的最大的元素的问题 例题 将循环展开成循环体的4个副本,假定R1 - R2(即数组的大小)初始时是32的倍数,即循环次数是4的倍数。消除明显的冗余计算,且不重用寄存器。

解答: 以下是经过合并DADDUI指令并消除在循环展开过程中重复的BNE操作之后的结果。注意,这里的R2必须被设定以保证32(R2)是最后四个元素的起始地址。

```
Loop:  L.D      F0,0(R1)
      ADD.D   F4,F0,F2
      S.D     F4,0(R1)      ;去掉 DADDUI & BNE
      L.D     F6,-8(R1)
      ADD.D   F8,F6,F2
      S.D     F8,-8(R1)    ;去掉 DADDUI & BNE
      L.D     F10,-16(R1)
      ADD.D   F12,F10,F2
      S.D     F12,-16(R1) ;去掉 DADDUI & BNE
      L.D     F14,-24(R1)
      ADD.D   F16,F14,F2
      S.D     F16,-24(R1)
      DADDUI  R1,R1,#-32
      BNE     R1,R2,Loop
```

这里我们省掉了三次转移操作和三次递减R1的操作。同时通过补偿load和store的地址,使得操作R1的DADDUI指令可以合并。这类优化并不像看上去那么简单,这里需要符号的替换和化简。

符号的替换和化简使我们可以重构表达式以合并常量,从而将像“ $((i+1)+1)$ ”这类表达式重写为“ $(i+(1+1))$ ”,进而简化为“ $(i+2)$ ”。这类优化是通过消除相关计算来进行的,我们将来在附录G中看到它的更一般形式。

进行调度之前,在展开的循环中每一个操作后面都跟随着一个相关操作,因而会造成停顿。在上述循环的执行过程中,每条LD指令之后需要1次停顿,每条ADDD指令之后需要2次停顿,DADDUI指令之后需要1次停顿,再加上发射指令所花费的14个时钟周期,该循环的执行一共要花费27个时钟周期,也就是说以每4个元素为一组,每组中的任意一个元素平均要花费6.75个时钟周期。但是我们可以通过有效地调度使其性能得到显著提升。通常情况下,循环展开是在编译过程的早期完成的,这使得优化器可以发现冗余计算并将其消除。

在实际程序中,我们通常无法确定循环的上限。假设上限为 n ,我们将循环展开,获得循环体的 k 个副本,这样我们得到的将是一对连续的循环,而不是单独的循环展开。其中第一个循环执行 $(n \bmod k)$ 次,其循环体与原始循环相同。第二个循环执行 (n/k) 次,其循环体是展开的循环。当 n 的值很大时,大部分执行时间将花费在展开的循环体上。

在上面的例子中,循环展开通过消除额外开销指令提高了性能,虽然这会使代码量显著地增加。如果我们将以上代码放在前面描述的流水线中进行调度执行,情况又会如何呢?

例题 使用时延情况如图2.2所示的流水线,写出上例中的循环展开经过调度后的执行情况。

```
解答: Loop:  L.D      F0,0(R1)
              L.D      F6,-8(R1)
```

L.D	F10, -16(R1)
L.D	F14, -24(R1)
ADD.D	F4, F0, F2
ADD.D	F8, F6, F2
ADD.D	F12, F10, F2
ADD.D	F16, F14, F2
S.D	F4, 0(R1)
S.D	F8, -8(R1)
DADDUI	R1, R1, #-32
S.D	F12, 16(R1)
S.D	F16, 8(R1)
BNE	R1, R2, Loop

这个循环展开的执行时间总共只有 14 个时钟周期, 或者说每个元素 3.5 个时钟周期, 既少于不经过调度也不展开时的 9 个时钟周期, 也少于在不展开的情况下进行调度的 7 个时钟周期。

与对原始循环进行调度的情况相比, 对循环展开后的代码进行调度会获得更显著的性能提升。这是由于循环展开可以通过调度来利用更多的计算将停顿减至最小。比如上述代码就没有停顿。这种调度方式要求识别 load 和 store 是不相关且可交换的。

循环展开和调度的小结

在这一章和附录 G 中, 我们将会看到各种各样的软件技术和硬件技术, 这些技术使我们可以利用指令级并行性的优势, 充分发挥处理器功能单元的潜能。这些技术的关键在于要确定指令的顺序在什么情况下是可以改变的, 以及怎样改变指令的顺序。前面的例子以直观的方式介绍了改变指令顺序的方法, 在实际应用中, 这些工作必须由编译器或硬件以系统的方式来完成。要最终获得展开的代码, 我们必须做出下面的决策和转换:

- 如果循环迭代之间是互不相关的, 则可以判定循环展开是有意义的, 维护循环的代码除外。
- 为不同的计算使用相同的寄存器会引起一些额外的限制, 需要使用不同的寄存器来避免这类限制。
- 消除额外的测试和转移指令, 调整循环终止和迭代代码。
- 如果来自不同循环体的 load 和 store 是互不相关的, 则可以确定循环展开中的 load 和 store 是可交换的。这需要对存储器地址进行分析, 以确定上述指令引用的不是相同的地址。
- 调度代码, 保持相关性, 以确保得到和原始代码相同的结果。

上述这些转换隐含着一个重要的共同点, 就是必须要确定指令间的相关性; 而在已知相关性的情况下, 必须确定如何改变和重组指令。

循环展开的效果受到三类因素的限制: 展开使额外开销降低、代码量的大小和编译器的限制。首先来考察循环开销的问题。当 4 次展开循环时, 指令间的并行度已足够使调度消除循环中的停顿。实际上, 在 14 个时钟周期中, 只有 2 个时钟周期是循环开销: 即维护循环索引的 DADDUI 和终止循环的 BNE。如果循环展开 8 次, 则额外开销将从最初的每次迭代 1/2 个时钟周期减小到 1/4 个时钟周期。

限制循环展开的第二类因素是代码量的增长。特别是在规模较大的循环中,当代码量的增长使指令 Cache 的缺失率增加时,这类问题就更加明显。

比代码量的增长更为严重的问题是,大规模地展开和调度循环可能会引起寄存器的短缺,这类由大规模代码段中的指令调度引起的副作用称为寄存器不足。为了增加指令级并行性,对代码的调度会增加活性变量的数目。而在大规模的指令调度后,想为所有的活性变量分配寄存器几乎是不可能的。虽然理论上转换后的代码应当在执行性能上拥有优势,但是寄存器的短缺却有可能使这种优势降低,甚至消耗殆尽。在不展开的情况下,调度的规模会受到转移的限制,因此很少出现寄存器不足的情况。但如果将展开和大规模调度相结合,寄存器不足就成为一个亟待解决的问题。特别是在多发射处理器中,由于要重叠执行更多的独立指令序列,寄存器不足带来的挑战会更为严峻。一般来说,在代码生成之前,复杂的高级转换带来的改进效果是很难衡量的,这些转换的使用已经显著地增加了现代编译器的复杂度。

为了增加可调度的直线代码段,循环展开是一种简单而有效的方法。循环展开这种转换方法对于各种处理器——从前面提到的简单流水线到超标量多发射结构,再到后面将要看到的 VLIW——都是适用的。

2.3 采用预测技术减小转移开销

由于需要通过转移冒险和停顿来保持控制相关,因此转移会降低流水线的性能。循环展开是减小转移冒险的方法之一;此外,我们还可以通过使用转移预测技术来减小转移引起的性能损耗。

转移预测既可以在编译阶段静态完成,也可以由硬件在执行阶段动态完成。在有些处理器中,转移行为在编译阶段是高度可预测的,静态转移预测既可以应用于这类处理器,也可以用来辅助动态转移预测。

静态转移预测

在附录 A 中,我们介绍了一种支持静态转移预测的系统结构特征,即延迟转移。在编译阶段准确地预测转移也有助于对数据冒险的调度。循环展开是另一类依靠转移预测对代码调度进行改进的技术的例子。

为了重组转移周围的代码以获得更高的性能,需要在程序编译阶段静态预测转移的行为。最简单的方法是预测转移总是被选中。这种方法的错误率等于转移不被选中的比率,对于 SPEC 程序来说这个比率平均为 34%。遗憾的是, SPEC 程序的错误率并不稳定,而是在 59%~9% 之间波动。

另一种技术通过分析转移的历史表现来进行转移预测,这种技术拥有更高的准确率。它的依据在于转移的表现通常遵循双峰分布,也就是说具体到某个转移,它或者被选中,或者不被选中。图 2.3 表明通过这种策略进行转移预测是成功的。在执行和收集历史信息时通常使用相同的输入数据,研究表明,如果在收集历史信息的过程中使用不同的输入数据,只会对这种预测的准确率产生微小的影响。

对于所有的转移预测机制来说,它的有效性取决于该机制的准确率和条件转移的频率(对 SPEC 来说该频率通常在 3%~24% 之间波动)。定点程序的预测错误率和转移频率通常偏高,这些是限制静态转移预测的主要因素。而当前的处理器大多采用动态预测技术,我们将在下一节中讨论动态转移预测。

静态转移预测

静态转移预测

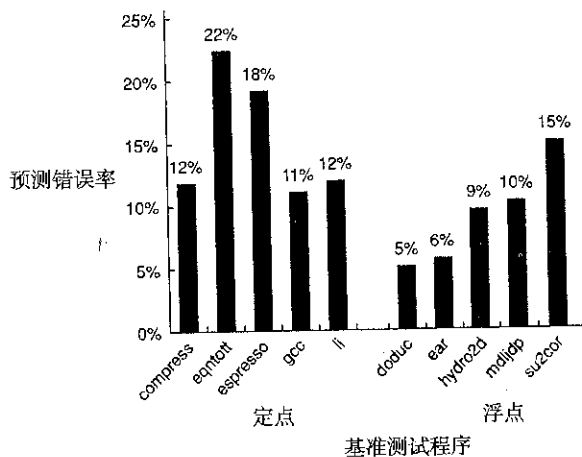


图 2.3 在 SPEC92 上使用基于历史信息的预测, 其预测错误率波动较大, 定点程序的预测错误率平均为 15%, 偏差为 5%, 而浮点程序的效果相对来说要好一些, 其预测错误率平均为 9%, 偏差为 4%。实际性能取决于预测准确率和转移频率, 后者通常在 3%~24% 之间波动

动态转移预测和转移预测缓存

动态预测最简单的方法是**转移预测缓存**或**转移历史表**。转移预测缓存是一小块由转移指令低地址索引的存储单元, 用来记录转移指令在最近的一次执行中是否被选中。这种方法是缓存中最简单的一种, 没有标志位, 而且只有当转移延迟高于计算目标 PC 所花的时间时才起作用。

实际上, 我们无法通过这样一块缓存来判断预测是否正确——因为另一条拥有相同低地址的转移指令也许会将这块缓存覆盖掉。不过这不是问题, 因为预测只是一个提示, 它被假设为是正确的, 并从预测的方向开始取指令。如果这个提示是错误的, 只需要对预测位求反并将其写回即可。

从效果上看, 这块缓存等同于一块每次都命中的 Cache, 我们将会看到, 它的性能取决于转移的预测频率及准确率。在分析性能之前, 先来看一个为提高准确率而做的虽小但却很重要改进。

简单的 1 bit 预测法在性能上有一点不足: 假设一个转移几乎总是被选中, 而当它偶尔未被选中时, 错误预测会使预测位空翻, 从而使预测错误两次, 而不是一次。

通常使用 2 bit 预测法来弥补这个缺陷。在这种预测法中, 仅当预测错误两次时才改变预测方向。图 2.4 所示为 2 bit 预测法的有限状态机。

转移预测缓存可以作为一个在 IF 流水阶段通过指令地址访问的专用 Cache 来实现, 也可属于指令 Cache 中的每一块, 随指令一起读取。如果经过译码后发现指令为转移指令, 并且预测转移将被选中, 则当确定 PC 后立刻从预测方向上开始取指令。否则仍继续按顺序取指令和执行。图 2.4 所示, 预测错误时改变预测位。

在实际应用中, 每个入口采用 2 bit 预测机制的转移预测缓存会获得怎样的准确率呢? 在 SPEC89 基准测试程序, 使用有 4096 个入口的转移预测缓存会获得 82%~99% 的准确率, 或 1%~18% 的错误率。以 2005 年的标准来看, 一个有 4K 个入口的缓存是比较小的, 可以通过更大缓存获得更高的准确率。

要想开发指令级并行, 转移预测的准确率是个关键问题。正如在图 2.5 中看到的, 对于转移率较高的定点程序, 其预测准确率要低于循环集中的科学计算程序。我们可以通过两种方法尝试解决这个问题: 增加缓存的容量, 增加每次预测所使用的预测方案的准确率。如图 2.6 所示,

4K个入口的缓存与一个无限大的缓存相比,在准确率上并没有明显的不同,至少对于SPEC这样的基准测试程序是这样。图2.6中的数据清楚地表明,缓存的命中率并不是主要的限制因素。正如在前面提到的,仅仅增加预测器的位数而不改变其结构所产生的影响是极其有限的。因此,需要研究提高每个预测器准确率的方法。

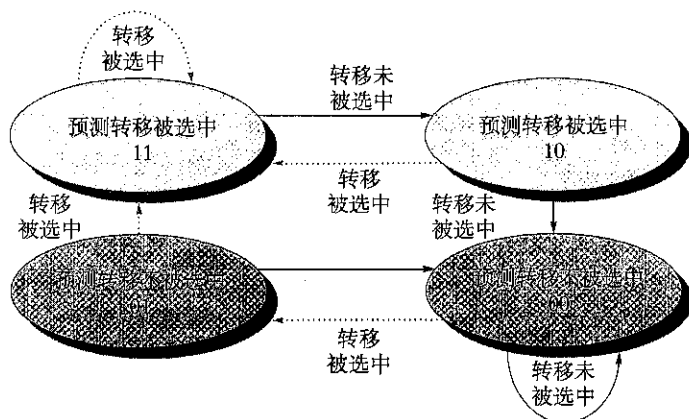


图2.4 2 bit预测方法状态机。对于有强烈倾向性的转移来说(大多数转移都属于这一类型),2 bit预测方法的错误率要比1 bit预测方法的低。在系统中,2 bit可以编码4种状态。而实际上,更一般的方法是为预测缓存中的每个入口使用一个 n bit的计数器,2 bit方法是这种方法的特殊形式。一个 n bit计数器的取值范围在0到 $2^n - 1$ 之间:当计数器的值大于或等于最大值($2^n - 1$)的一半时,预测转移将被选中;反之预测转移不被选中。研究发现,2 bit预测方法的效果几乎与 n bit预测方法的效果相同,因此大多数系统使用2 bit预测方法而不是更一般的 n bit预测方法。

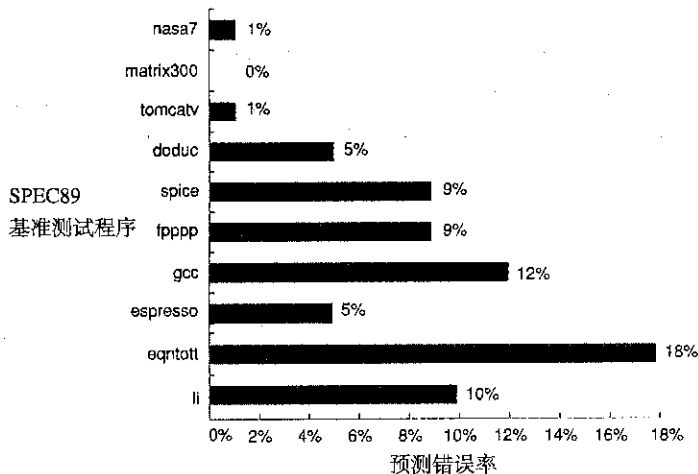


图2.5 一个有4096个入口的2 bit预测缓存对SPEC89基准测试程序的预测准确率。定点基准测试程序(gcc, espresso, eqntott和li)的预测错误率(平均为11%)要明显高于浮点程序的预测错误率(平均为4%)。即使忽略掉浮点内核(nasa7, matrix300和tomcatv),浮点基准测试程序的准确率仍然要比定点基准测试程序的高。以上以及本节中的其他数据,均源于对IBM Power系统结构的转移预测和对这些系统的代码优化研究,可参考Pan, So和Rameh[1992]。这些数据是针对老版本的SPEC基准测试程序子集的,而新的基准测试程序则更大,其表现要稍微差一些,这些在定点基准测试程序中表现得更加明显。

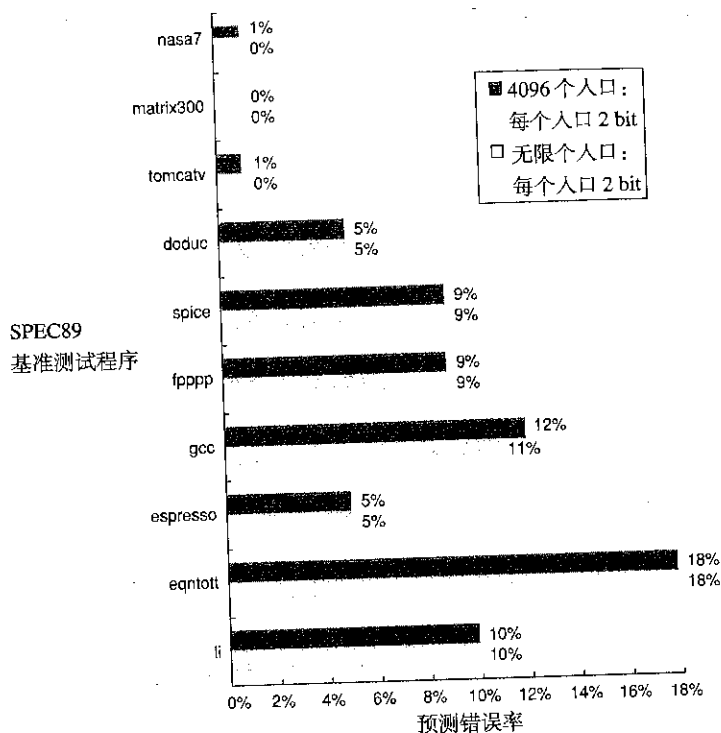


图 2.6 4096 个入口 2 bit 预测缓存的预测准确率与无限缓存对于 SPEC89 基准测试程序的预测精度的比较。尽管这些数据是针对老版本的 SPEC 基准测试程序子集的，而新版基准测试程序可能拥有 8K 个入口的缓存，需要匹配无限大的 2 bit 预测器，但它们在效果上几乎相当

相关转移预测

2 bit 预测通过分析转移在最近一次执行中的行为来预测其将来的行为。如果我们将其他转移在最近一次执行中的行为也考虑在内，而不是仅仅考虑将要预测的转移，也许会对提高预测的准确率有所帮助。考虑 SPEC 基准测试程序（其转移预测效果较差）的如下代码片断：

```
if (aa==2)
    aa=0;
if (bb==2)
    bb=0;
if (aa!=bb) {
```

以下是由上述代码片断产生的 MIPS 代码，假设 aa 和 bb 分别被分配给寄存器 R1 和 R2：

```

DAD0IU    R3,R1,#-2
BNEZ      R3,L1          ;branch b1  (aa!=2)
DADD      R1,R0,R0        ;aa=0
L1:       DADDIU    R3,R2,#-2
BNEZ      R3,L2          ;branch b2  (bb!=2)
DADD      R2,R0,R0        ;bb=0
L2:       DSUBU     R3,R1,R2 ;R3=aa-bb
BEQZ      R3,L3          ;branch b3  (aa==bb)
```

将这些转移标记为 b1, b2 和 b3。关键的一点是转移 b3 的行为与 b1 和 b2 的行为有关。就是说，如果 b1 和 b2 都未被选中（即条件都为真且 aa 和 bb 的值都为 0），由于 aa 和 bb 相等，则 b3 将被选中。而如果只对单一的转移行为进行预测，则无法捕捉到这个行为。

通过分析其他转移的行为进行转移预测的方法称为相关预测或双级预测。对一个给定的转移, 现有的相关预测在工作时将最近执行的转移的行为也考虑在内。例如, 一个(1, 2)预测器根据最近执行的1个转移的行为, 在2个2 bit转移预测器中选择一个, 做出对特定转移的预测。其一般形式为, (m, n) 预测器根据最近执行的前 m 个转移的行为, 在 2^m 个转移预测器中选择一个, 每个转移预测器为 n bit 预测器。相关预测的吸引力在于, 与2 bit方法相比, 它可以获得更高的准确率, 而只需要少量的额外硬件支持。

这种硬件的简单性源于, 最近执行的前 m 个转移的历史行为可以记录在一个 m bit 的移位寄存器中, 其中的每一位用来记录转移是否被选中。这样就可以通过将转移地址的低位和 m bit 的全局历史相拼接, 从而对预测缓存进行编址。例如, 在由64个入口组成的(2, 2)缓存中, 转移指令的低4位和表示最近的两个转移行为的2位拼接成一个6位地址, 可对64个计数器进行寻址。

同标准的2 bit预测方法相比, 相关转移预测器有哪些优势呢? 要客观地比较它们, 就必须使用相同数量的状态位。 (m, n) 预测器中的bit数目为

$$2^m \times n \times \text{转移地址所能选择的入口数目}$$

而一个没有全局历史记录 的2 bit 预测器只是一个简单的(0, 2)预测器。

例题 一个有4K个入口的(0, 2)转移预测器共有多少位? 一个有相同位数的(2, 2)预测器有多少个入口?

解答: 对于有4K个入口的预测器, 有

$$2^0 \times 2 \times 4K = 8K \text{ bit}$$

在一个预测缓存为8K bits 的(2, 2)预测器中有多少可供转移选择的入口?

$$2^2 \times 2 \times \text{转移可选择的预测入口的数目} = 8K$$

因此, 可供转移选择的入口的数目 = 1K。

图2.7比较了4K个入口的(0, 2)预测器和1K个入口的(2, 2)预测器的预测错误率。正如我们所看到的, 相关预测器在准确率上不仅高于状态位位数相同的2 bit 预测器, 而且通常也胜过有无限个入口的2 bit 预测器。

Tournament 预测器: 整体局部自适应预测器

相关预测的首要目标是要通过增加全局信息, 解决标准2 bit 预测只考虑局部信息而引起的在一些重要转移上预测失败的问题, 从而最终达到提高性能的目标。Tournament 预测器通过使用多个预测器——其中一个基于全局信息, 另一个基于局部信息, 通过一个选择器将二者结合——而将这种方法又向前推进了一步。Tournament 预测器以中等大小的预测位(8K~32K bit)获得更高的准确率, 同时可以有效地利用大量预测位。现有的Tournament 预测器为每个转移使用2 bit 饱和计数器, 通过分析哪个预测器(局部、全局或某种程度的混合)在最近的预测中效率最高, 在两种不同的预测器中做出选择。而2 bit 预测器只有在做出两次错误预测后才会改变预测的方向。

Tournament 预测器的先进性在于, 它可以为特定的转移选择正确的预测器, 而这一点对于定点基准测试程序来说十分关键。对一个SPEC 定点基准测试程序, 一个典型的Tournament 预测器为其选择全局预测器的概率大约为40%, 而对于SPEC浮点基准测试程序来说, 这个概率则只有不到15%。

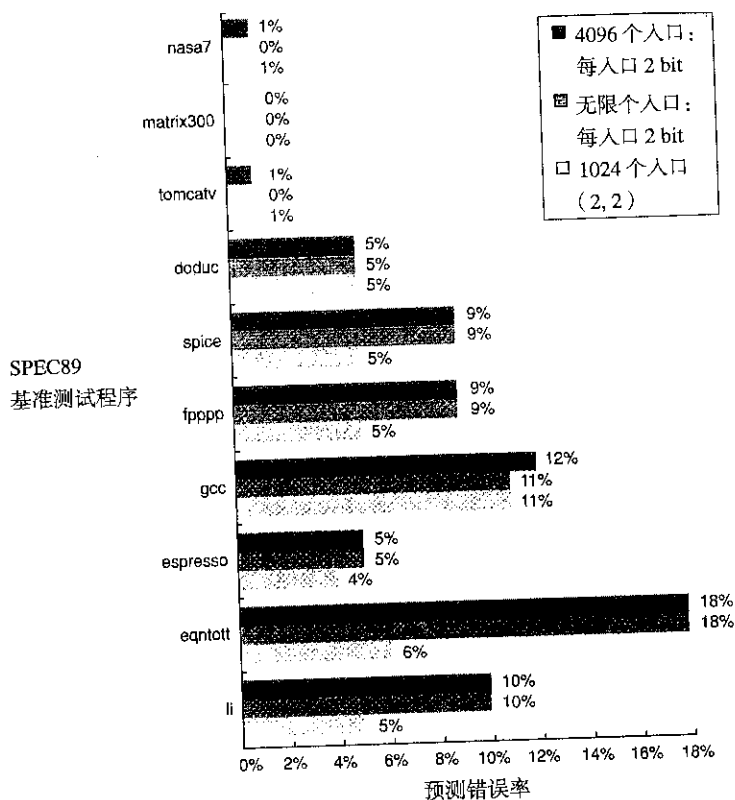


图 2.7 2 bit 预测器的比较。准确率最低的是 4096 位的不相关预测器，其次是无限个入口的不相关 2 bit 预测器，最后是有 2 bit 全局历史信息、总共 1024 个入口的 2 bit 预测器。尽管这些数据是由一个 SPEC 的老版本得到的，但是由新的 SPEC 基准测试程序得到的数据也会体现出相似的差异

图 2.8 所示为，当使用 SPEC89 作为基准测试程序时，三类预测器的性能（局部 2 bit 预测器、相关预测器和 Tournament 预测器）与所用位数之间的关系。正如我们前面所看到的，当预测位增长到一定数目后，局部预测器的性能不再有明显的改善。而相关预测器的性能仍有明显提高，Tournament 预测器的性能则比相关预测器的更好一些。在新版本的 SPEC 中也可以得到相似的结果，这种性能随预测位的增长而逐渐改善的情况会持续一段时间，直到预测位的数目达到一个更大的值。

2005 年，在处理器中使用大约 30K bit 的 Tournament 预测器是标准情况，比如 Power 5 和 Pentium 4。不过最先进的还是 Alpha 21264 中的预测器。21264 的 Tournament 预测器使用由局部转移地址索引的 4K 个 2 bit 计数器，在全局预测器和局部预测器中进行选择。全局预测器共有 4K 个入口，由最近执行的 12 个转移进行索引；其中的每一个入口为一个标准 2 bit 预测器。

局部预测器由两层组成。上层是由 1024 个 10 bit 入口组成的局部历史表；每一个 10 bit 入口对应这个转移最近 10 次的执行情况。就是说，如果这个转移连续被选中 10 次以上，则局部历史表中与该转移对应的 10 bit 入口的所有位均为 0。而如果这个转移有时被选中而有时不被选中，则 10 bit 入口的各位就是 0 和 1 相间的。这个 10 bit 历史信息最多可以预测 10 个转移，局部历史表中被选中的入口用来对一个表进行索引，这个表由 1K 个入口组成，每个入口为一个 3 bit 饱和计数器。该计数器用来进行局部预测。这种组合共使用了 29K bit，带来了更高的转移预测准确率。

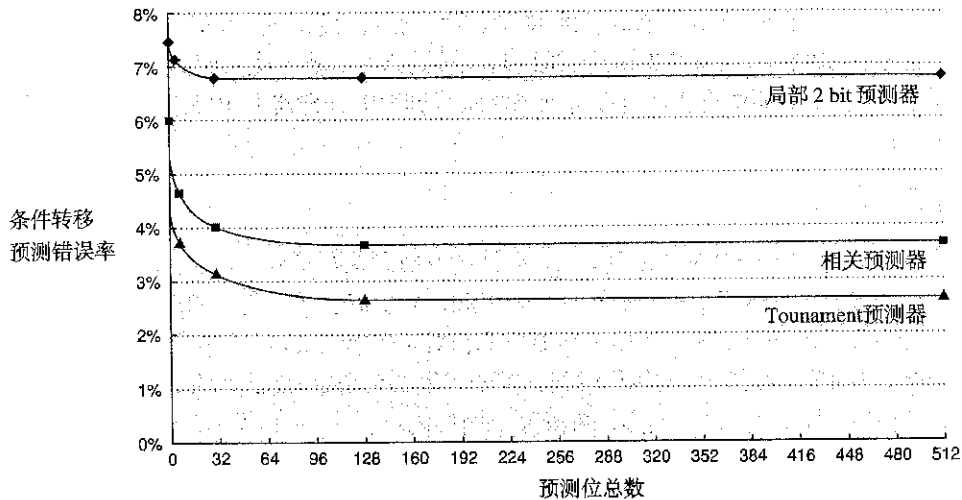


图 2.8 三种不同的预测器在运行 SPEC89 基准测试程序时，预测错误率同总位数之间的关系。这些预测器包括：局部 2 bit 预测器，对图中每一点的全局和局部信息使用结构优化的相关预测器，和一个 Tournament 预测器。这些数据是在运行一个较早版本的 SPEC 时得到的，而在新版本 SPEC 基准测试程序中也会呈现出类似的特性，预测错误率随着预测器尺寸的增大会趋于一个定值

为了检查预测对性能产生的影响，我们需要在了解预测准确性的同时了解转移频率，这是由于准确的预测在转移频率较高的程序中具有更重要的意义。比如，与更容易预测的浮点程序相比，SPEC 中定点程序的转移频率更高。对于 21264 的预测器，SPECfp95 基准测试程序每 1000 条指令中的预测错误不到 1，而 SPECint95 每 1000 条指令中的预测错误大约为 11.5。而相应的预测错误率则分别为，浮点程序不到 0.5%，定点程序大约 14%。

SPEC 的近期版本包含更大的数据集和代码量，因此会导致更高的错误率。而转移预测的重要性也随之增加。在 2.11 节中，我们将会讨论 Pentium 4 转移预测器在 SPEC2000 中的性能，我们将会看到，除了那些深度转移预测外，定点程序的转移预测错误率偏高仍然是一个棘手的问题。

2.4 采用动态调度克服数据冒险

一个简单的静态调度流水线负责取指令并将它发射出去，除非流水线中的指令与取到的指令之间存在数据相关，而且无法通过旁路技术和直接通路技术避免该数据相关（直传逻辑技术能够有效减少流水线延迟，从而使相关不会引起冒险）。如果有无法避免的数据相关，那么检测冒险的硬件将从使用相关结果的指令开始暂停流水线。停止取指令 and 发射指令的工作，直到相关被清除。

在本节中，我们将探讨动态调度技术，通过硬件对指令执行顺序进行重组，在保持数据流和异常行为的同时减少停顿。动态调度有以下优势：它可以处理一些在编译阶段无法预见的相关情况（如涉及存储器引用时），同时它简化了编译器的设计。也许最重要的是，它可以在等待时执行一些其他的代码，从而使处理器容忍像 Cache 缺失这样无法预见的延迟问题。同样重要的是，动态调度允许在别的流水线机器上编译的指令在不同的流水线上有效地运行。在 2.6 节中，我们将探讨硬件推测技术，它建立在动态调度的基础之上，在性能上具有明显的优势。我们将会看到，动态调度的这些优势是以硬件复杂度的显著增加为代价的。

Speculative

尽管处理器通过动态调度不能改变数据流,但是它会在相关性出现时尽力避免停顿。而由编译器执行的静态流水线调度(见2.2节)则试图将相关的指令分离,使它们不再引起冒险,从而将停顿减至最小。当然,对那些将要运行在有动态流水线调度的处理器上的代码来说,编译器流水线调度也是适用的。

动态调度: 概念

简单流水线技术的一个主要限制,是它们按序发射和执行指令: 指令以程序顺序发射,一旦指令在流水线中停顿,后续指令就无法再执行。因此,如果流水线中两条相近的指令之间存在相关,则会造成冒险并引起停顿。如果有多个功能单元,则这些单元会被闲置。如果指令*j*与正在流水线中执行的指令*i*相关,而指令*i*的执行时间又很长,那么在指令*i*完成、指令*j*可以执行之前,指令*j*之后的所有指令都必须暂停。例如,考虑如下的代码:

```
DIV.D    F0,F2,F4
ADD.D    F10,F0,F8
SUB.D    F12,F8,F14
```

在这个例子中,由于ADD.D相关于DIV.D,引起流水线停顿,因此指令SUB.D不能被执行;尽管SUB.D与流水线中的任意一条指令不相关。如果不要求指令以程序的顺序执行,则可以消除这种由冒险引起的对性能的影响。

在传统的五级流水线中,结构冒险和数据冒险可以在指令解码阶段被检测到: 当一条指令的执行不会引起冒险时,它会从已解决所有数据冒险的ID发射出去。

为了能使上例中的SUB.D指令开始执行,必须将发射过程分为两部分: 检查所有的结构冒险,等待数据冒险的消失。虽然指令的发射仍然采用按序的方式(即按程序顺序发射指令),但是我们要求指令在它的操作数可用时马上开始执行。这时流水线采取的是乱序执行的方式,这意味着指令的结束也是乱序的。

乱序执行可能会引起WAR和WAW冒险,而在五级定点流水线以及它的逻辑扩展——按序浮点流水线中,则不存在这个问题。考虑下面的MIPS浮点代码序列:

```
DIV.D    F0,F2,F4
ADD.D    F6,F0,F8
SUB.D    F8,F10,F14
MUL.D    F6,F10,F8
```

ADD.D和SUB.D之间存在反相关。如果流水线在ADD.D(等待DIV.D的执行)之前执行SUB.D,则该反相关特性将被破坏,导致WAR冒险。同样,为了避免改变输出相关,比如MUL.D对F6的写操作,必须处理WAW冒险。我们将会看到,这两种冒险都可以通过寄存器重命名来避免。

指令的乱序完成会使异常处理变得复杂。乱序执行的动态调度必须保护异常行为,以确保除严格按照程序顺序执行时出现的异常之外,不会出现新的异常。为保护异常行为,除非执行的是已知会产生异常的指令,否则动态调度的处理器必须保证指令的执行不会产生异常。后面的有关内容将介绍实现细节。

即使异常行为必须被保护,动态调度也会产生一些不精确异常。所谓不精确异常是指,当产生异常时,处理器状态与严格按照程序顺序执行时的处理器状态不同。不精确异常的发生有以下两种可能:

1. 在程序顺序中引起异常的指令执行之前,流水线提早完成了后面的指令。
2. 在程序顺序中引起异常的指令完成之前,流水线还没有完成前面的指令。

不精确异常会使出现异常后重新开始执行原先的指令变得更加困难。不过这一节不会研究这些问题，而是讨论在具有推测能力（见 2.6 节）的处理器环境中，提供精确异常的解决方案。浮点异常会使用一些其他的解决方案，这些内容将在附录 J 中介绍。

为了实现乱序执行，我们需要将简单五级流水线的 ID 流水阶段分割为以下两个部分：

1. 发射：译码指令，检测结构冒险。
2. 读操作数：等到不存在数据冒险时，开始读操作数。

取指令阶段在发射阶段之前，指令有可能被取到指令寄存器中，也有可能被取到等待指令序列中；随后，指令从寄存器或队列中被发射出去。在五级流水线中，EX 阶段在读操作数阶段之后。指令的执行可能会花费多个周期，这取决于具体的操作。

我们需要确定指令何时开始执行以及何时执行完成；这两个时刻之间就是指令的执行状态。如果流水线不允许多条指令同时执行，动态调度的主要优势将不复存在。并行执行多条指令需要处理器有多个功能单元，或多个流水线功能单元，或兼而有之。对于实现流水线控制，这两者——流水线功能单元和多个功能单元——在本质上是等价的，鉴于此，以后将假定处理器具有多个功能单元。

在动态调度流水线中，所有的指令在发射阶段都采用顺序的方式，即按序发射；而在第二个阶段，即读操作数阶段，则有可能产生停顿或旁路，即进入乱序执行。记分板技术允许在资源充足和没有数据相关的情况下乱序执行指令；CDC6600 发展了这种技术，记分板的命名就是在 CDC6600 之后。我们将在附录 A 中讨论记分板技术。这里我们将主要讨论一种更成熟的技术，即 Tomasulo 算法，同记分板相比，它在性能方面具有诸多优势。

用 Tomasulo 方法进行动态调度

IBM 360/91 的浮点功能单元使用一种更成熟的机制支持乱序执行。该机制由 Robert Tomasulo 提出，通过对操作数何时可用进行跟踪以减少 RAW 冒险，通过引入寄存器重命名减少 WAW 和 WAR 冒险。Tomasulo 方法的许多改进已应用于现代处理器中，它们的共同特点是通过跟踪指令的相关性使指令可以在操作数可用时立即开始执行，以及重命名寄存器，来避免 WAR 和 WAW 冒险。

IBM 的目标是能够从指令系统和为 360 系列计算机设计的编译器中获得更高的浮点性能，而不仅仅依赖为高端处理器设计的特殊编译器。360 系统结构只有四个双精度浮点寄存器，这限制了编译器调度的有效性；此外，IBM 360/91 具有较长的存储器访问和浮点处理延迟，如何解决这些问题是 Tomasulo 算法的设计目标。在这一节的最后，我们将会看到，Tomasulo 算法也可以支持循环中多次迭代的重叠执行。

下面我们结合 MIPS 指令系统的浮点单元和 load-store 单元解释该算法。360 体系中出现了寄存器-存储器指令，这是 MIPS 和 360 最主要的不同点。由于 Tomasulo 算法使用了一个 load 功能单元，因此我们不需要引入寄存器-存储器寻址方式。虽然 IBM 360/91 采用的是流水线功能单元而不是多个功能单元，但是我们在描述算法时仍然假设它有多个功能单元。两者之间只是一个简单的概念性扩展。

正如我们将会看到的，仅当指令的操作数可用时才执行指令以避免 RAW 冒险。而通过寄存器重命名，可以消除由名字相关引起的 WAR 和 WAW 冒险。寄存器重命名是指通过重命名所有的目标寄存器，包括指令序列中位置靠前的指令将要读和写的寄存器，从而消除相关，使乱序写不会影响那些与前面的操作数不相关的指令。

为了更好地理解寄存器重命名是怎样消除 WAR 和 WAW 冒险的，考虑下面这段代码序列，这段代码中隐含着 WAR 和 WAW 冒险：

```

DIV.D      F0, F2, F4
- ADD.D    F6, F0, F8
  S.D      F6, 0(R1)
- SUB.D    F8, F10, F14
- MUL.D    F6, F10, F8

```

在这个例子中，ADD.D 和 SUB.D 之间存在反相关，ADD.D 和 MUL.D 之间存在输出相关，这会引引起两类冒险：ADD.D 对 F8 的操作会引起 WAR 冒险，而由于 ADD.D 可能会晚于 MUL.D 完成，因此会引起 WAW 冒险。同时，这段代码中还存在三种真数据相关：DIV.D 和 ADD.D 之间，SUB.D 和 MUL.D 之间，以及 ADD.D 和 S.D 之间。

以上两种名字相关都可以通过寄存器重命名来消除。简单地说，假设存在两个临时寄存器 S 和 T。使用 S 和 T，上述序列代码可被重写为以下不包含相关性的代码序列：

```

DIV.D      F0, F2, F4
ADD.D      S, F0, F8
S.D        S, 0(R1)
SUB.D      T, F10, F14
MUL.D      F6, F10, T

```

此外，后续指令中所有对 F8 的引用必须被替换为寄存器 T。在这段代码中，所有的重命名处理都是由编译器静态完成的。而在随后的代码中使用 F8 需要有成熟的编译分析或硬件支持，这是由于在上述代码和以后使用 F8 的代码之间可能会存在转移指令的干扰。正如我们将要看到的，Tomasulo 算法可以处理跨转移重命名问题。

在 Tomasulo 算法中，寄存器重命名是通过保留站实现的，保留站为等待发射的指令保存操作数。其基本思想是：当操作数可用时，保留站马上取操作数并将其缓存，从而避免从寄存器中读操作数。此外，即将执行的指令指定保留站为其提供输入。最后，当对寄存器的后续写操作在执行过程中发生重叠时，只允许最后一个实际更新寄存器。在指令被发射后，它所需要的操作数所对应的寄存器将被重命名为保留站的名字，即寄存器重命名。

由于可以使用数量多于实际寄存器的保留站，因此，寄存器重命名甚至可以消除一些编译器无法消除的相关性。当我们探讨 Tomasulo 算法的组成部分时，我们将回到寄存器重命名的主题，并研究如何实现重命名以及寄存器重命名怎样消除 WAR 和 WAW 冒险。

使用保留站与使用集中式的寄存器堆相比有两个重要特点。首先，冒险检测和执行控制是分布的：一个单元中的指令何时可以开始执行是由该单元保留站所掌握的信息决定的。第二，结果将从缓存它们的保留站中直接传送给功能单元，而不是通过寄存器传送。这是通过一条公共结果总线实现的，它使得等待操作数的所有单元可以同时取到该操作数（在 360/91 中称为公共数据总线，或 CDB）。在有多条执行单元且一个时钟周期内可发射多条指令的流水线中，需要一条以上的结果总线。

图 2.9 所示为基于 Tomasulo 算法的处理器的基本结构，包括浮点单元和 load-store 单元；而执行控制表未表示出来。每个保留站保存一条已经被发射并等待执行的指令，如果指令所需要的操作数已经被计算出来，那么保留站需要保存该操作数，否则要保存将要提供该操作数的保留站的名字。

load 缓存和 store 缓存保存从存储器中读出或即将要保存到存储器中去的数据或数据地址，load 缓存和 store 缓存执行的操作几乎与保留站相同，因此以后我们只在需要时才会区分它们。浮点寄

序列，这 寄存器通过一对总线和功能单元相连，而通过一条单独的总线和 store 缓存相连。从功能单元和存储器中得到的所有结果被送往公共数据总线。通过公共数据总线，结果可以被送往除 load 缓存外的任何地方。所有的保留站均设置有标签域，用于流水线控制。

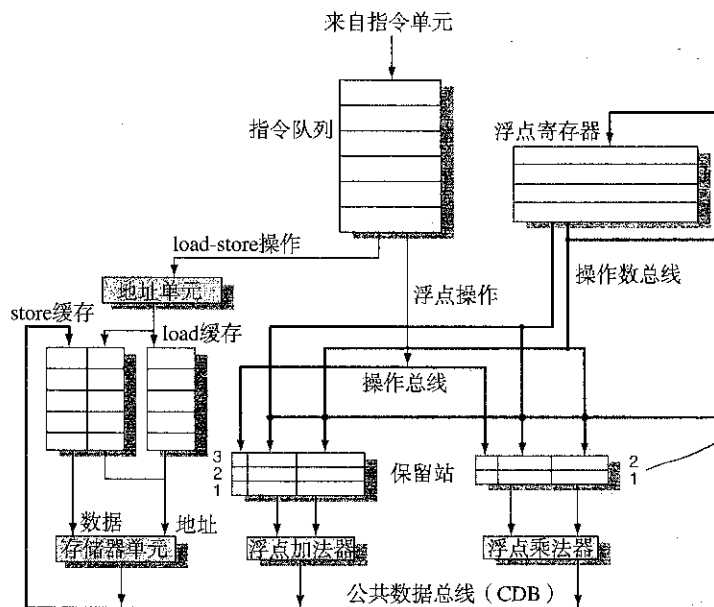


图 2.9 使用 Tomasulo 算法的 MIPS 浮点单元的基本结构。指令从指令单元送往指令队列，在指令队列中，指令按 FIFO 顺序发射。保留站中包括操作数和运算符，以及用来检测和解决冒险的信息。load 缓存具有三个功能：保存有效地址的各个部分直至计算完成，等待将要访问存储器的 load 指令，为已经完成且正在等待 CDB 的 load 指令保留结果。与之相似，store 操作也具有三个功能：保存有效地址的各个部分直至计算完成，为正在等待数据值的 store 指令保存目标存储器地址，在存储器单元可用之前保留地址和等待写回的值。来自浮点单元或 load 单元的所有结果都被送往 CDB，这些结果将被送往浮点寄存器文件、保留站和 store 缓存。浮点加法器执行加减运算，浮点乘法器执行乘除运算。

在介绍保留站和算法的细节之前，先让我们来分析一下指令运行的几个阶段。尽管每条指令花费的时钟周期不同，但总体上可分为 3 个阶段：

1. **发射**：从指令队列中取到下一条指令，其中指令队列按 FIFO 顺序维护，以保证正确的数据流。如果有匹配的空闲保留站，并且指令的操作数的值也保存在寄存器中，则将指令和操作数的值一起发射到该保留站中。如果没有空闲保留站，则说明发生了结构冒险，指令会被停顿，直至出现可用的保留站或缓存。如果操作数不在寄存器中，则需要跟踪将要生产该操作数的功能单元。寄存器重命名在这一步进行，以消除 WAR 和 WAW 冒险（在动态调度处理器中，这一阶段有时也称为指派阶段）。
2. **执行**：如果有一个或多个操作数处于不可用状态，则监视公共数据总线，等待这些操作数被计算出来。当一个操作数可用时，该操作数将被放入等待它的保留站中。当指令所需的所有操作数都已就绪时，该指令将在相应的功能单元中执行。这里，我们通过操作数可用之前延迟指令的执行，避免了 RAW 冒险（一些动态调度处理器将这一步称为“发射”，但是我们在这里将它命名为“执行”，这种命名方法与最早的动态调度处理器 CDC 6600 一致）。

需要注意的是,同一功能单元的多条指令可能会在同一个时钟周期内就绪。尽管相互独立的功能单元可以在同一个时钟周期内执行不同的指令,但是对某个特定的功能单元来说,如果有多条指令在同一个时钟周期内就绪,则该功能单元将被迫在就绪指令间做出选择。浮点保留站在这种情况下可以任意选择;但是对于 load 和 store 指令来说,情况要复杂得多。

load 指令和 store 指令需要分两步执行。第一步是当基址寄存器可用时计算有效地址,之后该有效地址将被置于 load 或 store 缓存中。当存储器单元可用时,load 缓存中的 load 指令将立即被执行。而 store 缓存中的 store 指令在被送往存储器单元之前,则要等待被保存的值。这样一来,通过计算有效地址,load 指令和 store 指令得以按照程序顺序执行,不久我们将会看到,这样做有助于避免访问存储器的冒险。

为了保护异常行为,在程序顺序中的所有前序转移完成之前,任何指令都不能开始执行。这个约束可以确保在执行过程中引起异常的指令会被执行。在使用转移预测的处理器中,这意味着处理器必须知道转移预测是正确的,才会允许转移后面的指令开始执行。如果处理器记录到异常情况的发生,但是异常并没有实际出现,则指令可以开始执行,并且不必停顿,直到进入写结果阶段。

推测技术提供了一种更为灵活和更为完整的方法来处理异常,至于推测技术是怎样处理异常的,我们将在后面进行讨论。

3. 写结果:当结果就绪时,需要将其写到 CDB 上,并由此送往等待它的寄存器和保留站(包括 store 缓存)。store 指令被缓存在 store 缓存中,直到将要保存的值和保存地址都可用为止,之后,当存储器单元闲置时,该结果将被立刻写往存储器。

检测和消除数据冒险的数据结构被附在保留站、寄存器文件以及 load 缓存和 store 缓存中,根据所附对象的不同,其携带的信息也略有不同。从本质上讲,这些标签就是用来支持重命名的虚拟寄存器名字的扩展集。在这个例子中,标签字段是一个 4 位容量的结构,它表示 5 个保留站中的某一个或 5 个 load 缓存中的某一个。我们将会看到,这样的效果等价于设定了 10 个结果寄存器(而 360 系统结构只包含 4 个双精度寄存器)。在一个拥有更多寄存器的处理器中,我们甚至希望通过重命名得到更多的虚拟寄存器。而标签字段指明了包含将要产生源操作数的指令的保留站。

当指令已经被发射且正在等待源操作数时,标签字段将指向包含将产生源操作数指令的保留站号。而像 0 这样的无效值,则表明操作器已经在寄存器中就绪。由于保留站的数量多于实际的寄存器,因此可以通过使用保留站号进行重命名以消除 WAW 和 WAR 冒险。Tomasulo 方法将保留站作为虚拟寄存器的扩展来使用,而其他的方法则可能使用有额外寄存器的寄存器集或者像重排序缓存这样的结构,这些内容我们将在 2.6 节中进行介绍。

Tomasulo 方法以及随后将要介绍的支持推测的方法,都是采用总线(即 CDB)广播结果的方式,由保留站监听。这种方法实现了静态流水线调度中直接通路和旁路技术的功能。而在动态调度方法中,由于结果及其用途的匹配只有在写结果阶段才能进行,因此,为了达到同样的目标,动态调度方法需要在源和结果之间增加一个时钟周期的时延。所以,与功能单元生产结果相比,动态调度流水线中生产指令和消费指令之间的有效时延至少要多花费一个周期。

在描述 Tomasulo 机制的过程中,使用了记分板机制中的术语,而未引入 IBM 360/91 使用的新术语。需要注意的是,Tomasulo 算法中的标签是指向将要生产结果的缓存和功能单元的;当指令发射到保留站中后,寄存器的名字将被丢弃。

每个保留站有以下七个字段:

- Op: 在源操作数 S1 和 S2 上所做的运算。

- Qj, Qk: 将要产生相应源操作数的保留站; 0 表示源操作数已经在 Vj 或 Vk 中就绪, 或表示该操作数是不必要的 (在 IBM 360/91 中称这些字段为 SINK 单元和 SOURCE 单元)。
- Vj, Vk: 源操作数的值。对于每个操作数来说, V 字段或 Q 字段中只有一个是有效的。对于读取指令, Vk 字段用来保存偏移量字段 (在 IBM 360/91 中称这些字段为 SINK 和 SOURCE)。
- A: 为计算 load 指令和 store 指令的存储器地址保存信息。初始时, 指令的立即数字段被保存在这里; 在地址计算结束后, 这里保存有效地址。
- Busy: 表明保留站及其相关功能单元已被占用。

寄存器文件有一个字段 Qi:

- Qi: 如果某个运算的结果应当被保存在这个寄存器中, 则 Qi 为包含该运算的保留站的序号。如果 Qi 的值为空 (或 0), 则表示当前没有以该寄存器为目标的活动指令, 这意味着寄存器的内容就是结果的值。

Load 缓存和 store 缓存各有一个字段 A。当第一步执行完成后, 该字段将被用来保留有效地址。

在下一节的开始, 我们将首先用一些例子来说明 Tomasulo 算法是如何工作的, 之后将对算法的细节进行介绍。

2.5 动态调度: 举例和算法

在详细讨论 Tomasulo 算法之前, 让我们首先来看几个例子, 这将有助于我们了解 Tomasulo 算法是如何工作的。

例题 写出下列代码在第一个 load 指令完成并写回结果时的信息表。

1.	L.D	F6, 32(R2)	
2.	L.D	F2, 44(R3)	
3.	MUL.D	F0, F2, F4	10
4.	SUB.D	F8, F2, F6	
5.	DIV.D	F10, F0, F6	40
6.	ADD.D	F6, F8, F2	

解答: 图 2.10 给出了三个表的信息。指令名 add, mult 和 load 的后缀数字表示保留站的标签数字。Add1 是来自第一个加法单元的结果标签。此外, 我们还加入了一个指令状态表, 它有助于对算法的理解, 但并不是实际硬件的一部分。当指令被发射后, 运算的状态被保存在保留站中。

与早期的简单方法相比, Tomasulo 算法的主要优势在于: (1) 分布式冒险检测以及 (2) 消除 WAW 和 WAR 冒险停顿。

第一个优势来自于分布式保留站和公共数据总线 (CDB) 的使用。如果有多条指令在等待同一个结果, 并且所有指令已经得到了所需的其他操作数, 则在这种情况下, 通过在 CDB 上广播结果, 所有指令可以同时被激活。而如果使用集中式的寄存器文件, 那么单元就只能在寄存器总线可用时, 从寄存器中读取所需的结果。

第二个优势, 即消除 WAW 和 WAR 冒险, 来源于使用保留站重命名寄存器, 以及在操作数可用时, 及时将操作数保存在保留站中。

例如, 在上面的代码中, 尽管存在涉及 F6 的 WAR 冒险, 但是 DIV.D 和 ADD.D 还是被同时发射。有两种方法可以消除这个冒险。第一, 如果为 DIV.D 提供操作数的指令已经完成, 则由 Vk 保

存该结果，从而使DIV.D的执行可以独立于ADD.D（本例中使用的就是这种方法）。第二，如果L.D还未完成，Qk将指向Load1的保留站，从而使指令DIV.D独立于ADD.D。这样一来，不论使用的是哪种方法，ADD.D都可以被发射，并开始执行。此后，任何对DIV.D结果的使用都将指向保留站，从而使ADD.D在不影响DIV.D的情况下，完成它的执行并将结果保存到寄存器中。

指令		指令状态						
		发射	执行	写结果				
L.D	F6,32(R2)	√	√	√				
L.D	F2,44(R3)	√	√					
MUL.D	F0,F2,F4	√						
SUB.D	F8,F2,F6	√						
DIV.D	F10,F0,F6	√						
ADD.D	F6,F8,F2	√						

保留站							
名字	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	yes	Load					45 + Regs[R3]
Add1	yes	SUB		Mem[34 + Regs[R2]]	Load2		
Add2	yes	ADD			Add1	Load2	
Add3	no						
Mult1	yes	MUL		Regs[F4]	Load2		
Mult2	yes	DIV		Mem[34 + Regs[R2]]	Mult1		

寄存器状态								
字段	F0	F2	F4	F6	F8	F10	F12	... F30
Qi	Mult1	Load2		Add2	Add1	Mult2		

图 2.10 在所有指令被发射后，保留站和寄存器标签的状态，这时只有第一条load指令完成了操作，并且将结果写向CDB。第二条load指令已经完成了计算有效地址的运算，正在等待存储单元。我们用 Regs[] 数组表示寄存器文件，用 Mem[] 数组表示存储器。注意，一个操作数在任何时候均可以由Q字段或V字段具体确定。加法指令在WB阶段有一个WAR冒险，它已经被发射，并且应当在DIV.D初始化之前完成

不久我们将会讨论消除WAW冒险的问题，在此之前，先来分析上述例子是怎样继续执行的。在这个例子以及本章后面的几个例子中，假设加法运算需花费2个时钟周期，乘法运算需花费6个时钟周期，除法运算需花费12个时钟周期。

例題 当上述代码执行到MUL.D并准备写回结果时，写出各状态表的信息。

解答：图 2.11 分别给出了三个表的信息。注意，当DIV.D的操作数被复制时，ADD.D已经完成了，因此克服了WAR冒险。即使读取F6的指令被延迟，对F6的加运算也不会引起WAW冒险。

指令		指令状态		
		发射	执行	写结果
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	√
MUL.D	F0,F2,F4	√	√	
SUB.D	F8,F2,F6	√	√	√
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√	√	√

保留站							
名字	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	no						
Add1	no						
Add2	no						
Add3	no						
Mult1	yes	MUL	Mem[45 + Regs[R3]]	Regs[F4]			
Mult2	yes	DIV		Mem[34 + Regs[R2]]	Mult1		

寄存器状态									
字段	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1					Mult2			

图 2.11 仅剩乘法指令和除法指令未完成

Tomasulo 算法：细节

图 2.12 所示为一条指令必须经过的检查和步骤。前面曾经提到，在进入独立的 load 或 store 缓存之前，load 指令和 store 指令需要经过一个功能单元计算有效地址。接下来，load 指令需要访问存储器并进入写结果阶段，将从存储器读出的值送往寄存器文件或正在等待的保留站。而 store 指令需要将结果写向存储器，在写结果阶段完成指令的执行。注意，不论目标是寄存器还是存储器，所有的写操作都是发生在写结果阶段的。这个约束简化了 Tomasulo 算法，同时，对于用推测技术扩展的 Tomasulo 算法来说，这个约束也很关键。

Tomasulo 算法：一个基于循环的例子

为了更好地理解寄存器重命名在消除 WAW 和 WAR 冒险中的作用，我们来考察下面的代码，这段代码将数组中的每个元素与 F2 中的标量相乘：

```

Loop:   L.D      F0,0(R1)
        MUL.D   F4,F0,F2
        S.D     F4,0(R1)
        DADDIU  R1,R1,-8
        BNE    R1,R2,Loop; branches if R1≠R2

```

如果预测转移被执行,那么使用保留站技术可以使这个循环中的多条指令同时开始执行。这是在不改变代码的情况下实现的——实际上,循环由硬件动态展开,将通过重命名获得的保留站作为附加寄存器。

指令状态	需要满足的操作	动作或记录工作
发射 浮点操作	保留站 r 空	<pre> if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0}; RS[r].Busy ← yes; RegisterStat[rd].Q ← r; </pre>
读取或保存指令	缓存 r 空	<pre> if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; RS[r].A ← imm; RS[r].Busy ← yes; </pre>
仅 load 指令		RegisterStat[rt].Qi ← r;
仅 store 指令		<pre> if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rs].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0}; </pre>
执行	(RS[r].Qj = 0) and (RS[r].Qk = 0)	对 Vj 和 Vk 中的操作数计算结果
浮点操作读		
读取 - 保存	RS[r].Qj = 0 & r 是读取 -	RS[r].A ← RS[r].Vj + RS[r].A;
步骤 1	保存队列的头	
读取步骤 2	读取步骤 1 完成	从 Mem[RS[r].A]中读
写结果	r 上的执行完成	<pre> ∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0}); </pre>
浮点操作	&CDB 就绪	<pre> ∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ← 0}); </pre>
或		<pre> ∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ← 0}); </pre>
读取		RS[r].Busy ← no;
保存	r 上的执行完成 & RS[r].Qk = 0	<pre> Mem[RS[r].A] ← RS[r].Vk; RS[r].Busy ← no; </pre>

图 2.12 算法的步骤及每步需要满足的条件。对于正在发射的指令, rd 是目标寄存器号, rs 和 rt 是源寄存器号, imm 是符号扩展立即数字段, r 是分配给指令的保留站或缓存。RS 是保留站的数据结构。由浮点单元或读取单元返回的值称为结果。RegisterStat 是寄存器状态数据结构(寄存器文件用 Reg[] 表示)。当指令发射到保留站或缓存中后,目标寄存器将自己的 Qi 字段设为该保留站或缓存的序号。如果操作数已经在寄存器中就绪,则将操作数保存在 V 字段。否则用 Q 字段表示将要产生源操作数值的保留站。指令在保留站中等待两个操作数就绪, Q 字段为 0 时表示操作数就绪。当指令已经发射,或与该指令相关的指令已经执行完成并写回结果时, Q 字段将被置为 0。当指令执行完成并且 CDB 可用时,指令将写回结果。字段 Qj 或 Qk 与完成指令的保留站相同的所有缓存、寄存器和保留站,通过 CDB 取得数据并设置 Q 字段,使其表示已经收到了值。这样, CDB 可以在一个时钟周期内将结果广播到多个目标中,如果等待的指令已经得到了所需的操作数,则可以在下一个时钟周期同时开始执行。在执行过程中, load 指令要经过两个阶段,而由于要等待要保存的值,因此 store 指令在写结果阶段的表现会略有不同。需要注意的是,为了保护异常行为,如果程序顺序中较早的转移指令尚未完成,则指令不能够执行。这是由于程序顺序的信息在发射阶段后会丢失,为了实现这个约束,通常采取在流水线中有等待转移的情况下阻止指令离开发射阶段的方法。在 2.6 节中,我们将看到推测技术是如何消除这种限制的

开始执行。这是假设循环中两次连续迭代内的所有指令已经被发射,而所有的浮点读写操作或运算都还没有完成的保留站作图2.13所示为这个时刻的保留站和寄存器状态表,以及load和store缓存的状态(忽略定点ALU,假设预测转移被选中)。这时,在乘法运算可以在4个时钟周期内完成的前提下,系统将保持有两个循环体在流水线中运行的状态,其CPI可以接近1.0。在达到稳定状态之前,需要花一个时钟周期处理其他的循环体。这要求投入更多的保留站保存正在执行的指令。我们在这一章后面将会看到,当采用多指令发射技术时,Tomasulo算法可以在一个时钟周期内执行一条以上的指令。

		指令状态			
		发射	执行	写结果	
i}	j} ← 0);	F0,0(R1)	1	√	√
j} ← 0);		F4,F0,F2	1	√	
		F4,0(R1)	1	√	
		F0,0(R1)	2	√	√
i}		F4,F0,F2	2	√	
j} ← 0);		F4,0(R1)	2	√	

		保留站					
		Busy	Op	Vj	Vk	Qj	Qk
ad1	yes	Load					Regs[R1] + 0
ad2	yes	Load					Regs[R1] - 8
ad1	no						
ad2	no						
ad3	no						
ad1	yes	MUL			Regs[F2]	Load1	
ad2	yes	MUL			Regs[F2]	Load2	
ad1	yes	Store		Regs[R1]			Mult1
ad2	yes	Store		Regs[R1] - 8			Mult2

		寄存器状态							
		F0	F2	F4	F6	F8	F10	F12	... F30
Load2									
Mult2									

图2.13 指令未完成时的两个活动的循环迭代。乘法器保留站中的项表明load为源。store保留站表明乘法目标是存储值的源

在访问不同地址的前提下,load和store指令可以安全地乱序执行。而如果load指令和store指令访问相同的地址,则会发生下面两种情况中的一种:

- 如果在程序顺序中load指令在store指令之前,则交换它们的执行顺序将导致WAR冒险。
- 如果在程序顺序中store指令在load指令之前,则交换它们的执行顺序将导致RAW冒险。

同理,交换两个访问相同地址的store指令将导致WAW冒险。

因此,为了在某一给定时刻判定一条 load 指令是否可以被执行,处理器必须检查在程序顺序中是否有位于该指令之前且与该指令有相同访存地址的 store 指令尚未完成。同理,store 指令也必须等待那些与它有共同的访存地址且在程序顺序中在它之前的 load 或 store 指令。我们将在 2.9 节中讨论消除这种约束的方法。

要想检测这种冒险,处理器必须计算出与早期的存储器运算有关的所有地址。为了保证处理器获得所有这类地址,一种简单的优化方法是按程序顺序计算有效地址(实际上,我们只需要保持 store 指令和其他存储器引用之间的相对顺序;就是说,load 指令可以自由地重新排序)。

下面首先来考虑 load 指令的情形。如果按照程序顺序计算有效地址,那么我们就可以通过检查所有工作状态下的 store 缓存的 A 字段来检测是否存在地址冒险。如果 load 的地址与某一工作状态下的 store 缓存的地址吻合,那么,在冒险 store 指令完成之前,这条 load 指令将不会被送往 load 缓存(有些实现方法将 store 要保存的值直接旁路给 load 指令,以减小 RAW 冒险的延迟)。

store 操作与此类似,不同的是,由于与 store 指令和 load 指令都有可能发生冒险,并且不能重新排序,因此处理器必须对 load 缓存和 store 缓存都进行冒险检测。

在上一节中我们曾经提到过,在转移预测足够准确的前提下,动态调度的流水线可以获得很高的性能。而 Tomasulo 算法的主要缺陷是它的复杂性,它需要大量的硬件投入。特别是,每个保留站都要求有一个相关联的 Cache,这需要复杂的控制逻辑支持。而且使用单独的 CDB 也会限制性能。尽管可以补充 CDB,但是每条 CDB 都必须与每个保留站维持交互,同时还必须为每条 CDB 配置相关的标签匹配硬件。

Tomasulo 算法运用了两种不同的技术:寄存器重命名技术和源操作数缓存技术。源操作数缓存技术用来解决当操作数在寄存器中就绪时出现的 WAR 冒险。我们后面将会看到,通过对寄存器早期数据的引用消失之前缓存结果,结合寄存器重命名技术,可以消除 WAR 冒险。在讨论硬件推测时将用到这种技术。

在 360/91 之后,Tomasulo 算法曾经被搁置了几年,但是自 20 世纪 90 年代开始,它又在多发射处理器中得到了广泛应用,这主要有以下原因:

1. Tomasulo 算法可以获得高性能,而不需要编译器将代码编译成针对某种流水线的特定格式,这对应用来说是个很有价值的特性。
2. 尽管 Tomasulo 算法是在 Cache 出现前设计的,但是克服 Cache 内在的不可预见延迟,已经成为动态调度的目标之一。乱序执行使处理器可以在 Cache 缺失的情况下继续执行指令,从而避免或部分减小 Cache 缺失的代价。
3. 由于处理器的发射能力变得越来越强大,并且处理器的设计会涉及到调度困难代码(比如大多数非数值代码)的性能问题,这些都使得寄存器重命名及动态调度技术变得越来越重要。
4. 由于动态调度是推测技术的重要组成部分,因此它与硬件推测技术一起,在 20 世纪 90 年代中期得到了广泛应用。

2.6 基于硬件的推测

当我们试图进一步开发指令级并行时,维护控制相关性便成为了一个严重的负担。转移预测技术减少了直接由转移引起的停顿,但是要想使处理器在一个时钟周期内执行多条指令,仅靠准确的预测恐怕无法使我们获得期望的指令级并行度。为了保持最高性能,一个宽发射处理器可能需要每个时钟周期都执行一条转移指令。因此,要想进一步开发并行度,就必须克服控制相关带来的问题。

我们可以通过推测转移的结果,并按照推测正确的情况执行指令,以达到克服控制相关的目的。这个机制对基于动态调度的转移预测做了一个细微但很重要的扩展。在实践中,假定转移预测的结果总是正确的,并按照预测的结果取指令、发射指令并执行指令;而在相同的情况下,动态调度仅仅获取和发射这些指令。当然,需要一种机制来处理推测失败时出现的问题。附录G中讨论了几种通过编译器支持推测的机制。在这一节中,我们将主要探讨硬件推测技术,该技术是动态推测技术的扩展。

基于硬件的推测综合了以下三种思想:通过动态转移预测选择要执行的指令,通过推测技术允许指令在控制相关消除之前开始执行(能够消除错误推测序列的影响),通过动态调度处理几个不同的基本块之间的调度(作为比较,没有推测的动态调度只能在基本块之间实现部分重叠,因为后继基本块中的指令只能在转移解决之后才可以开始执行)。

基于硬件的推测技术根据预测的数据流选择何时执行指令。这种执行程序的方法本质上是执行数据流:一旦运算所需的操作数就绪,就立即开始执行。

为了扩展Tomasulo算法并使其支持推测技术,我们必须将指令结果的旁路操作(推测执行指令需要指令结果的旁路操作)从实际的指令完成中分离出来。通过这种分离,我们可以允许指令将它的执行结果旁路给其他指令,而在确定而不是推测指令的执行之前,不允许做任何更新。

使用旁路指令结果的值与执行一个推测的寄存器读操作类似,这是由于在确定推测结果是否正确之前,我们无法知道读源寄存器的指令是否提供了正确的结果。如果能够确定对指令的推测是正确的,则我们将允许该指令更新寄存器或存储器;称指令执行过程中的这个额外步骤为指令提交。

实现推测技术的一个关键思想是允许指令乱序执行,但是要求指令必须按序提交,并且在指令提交之前阻止所有不可恢复的动作(比如更新状态或产生异常)。因此,当使用推测技术扩展动态调度时,我们必须将指令的完成与指令提交区分开来,因为指令可能在提交之前已经完成。在指令的执行过程中增加这个提交阶段需要一组硬件缓存的支持,使用这些缓存保存已经执行完成但还没有提交的指令执行结果。我们称这些硬件缓存为重排序缓存,也被用来在推测执行的指令之间传递结果。

重排序缓存(ROB)提供了附加的寄存器,这种方法与Tomasulo算法通过保留站扩展寄存器集类似。在指令运算完成到指令提交的这段时间里,ROB为指令保存结果。即在这段时间内,ROB是指令的操作数源,这与保留站在Tomasulo算法中提供操作数的做法是类似的。ROB与保留站的重要区别是:在Tomasulo算法中,当指令完成写结果的操作后,所有的后继指令都将从寄存器文件中读取结果。而在推测技术中,只有在指令提交之后寄存器文件才会被更新(明确已知指令应当被执行);即在指令执行完成到指令提交的这段时间内,将由ROB来提供操作数。ROB类似于Tomasulo算法中的store缓存,为简化起见,可以将store缓存的功能并入到ROB中。

ROB中的每一个入口都包含4个字段:指令类型、目标字段、值字段和就绪字段。指令类型字段表明指令是转移运算(没有目标结果)、store操作(以存储器地址为目标)还是寄存器运算(ALU运算或load操作,以寄存器为目标)。目标字段提供寄存器序号(对于load和ALU运算)或存储器地址(对于store),指令将把结果写向目标字段指向的寄存器或存储器地址。值字段用来在指令提交之前,保存指令执行结果的值。我们不久就会看到一个ROB的实例。最后,就绪字段表示指令已经完成了它的执行,并且其结果已经可用。

图2.14所示为含有ROB的处理器硬件结构。ROB中包容了store缓存。Store指令仍然分两步执行,但是其第二步将用来执行指令提交。尽管ROB替代了保留站的重命名功能,但是在指令发射到指令开始执行的这段时间里,仍然需要一个空间来缓存操作(及操作数)。这个功能仍然由

入口的序号
记录在保留站
ROB 仅用

视 CDB, 直至其广播结果时再更新 ROB 入口的值字段。为简化起见, 我们假设这些只在写结果阶段发生; 稍后我们将讨论如何弱化这一要求。

4. 提交: 这是完成指令的最后一个阶段, 这之后只有指令的结果仍被保留 (有的处理器称这个提交阶段为“完成”或“毕业”)。在提交指令时有三种不同的动作序列, 采用哪种动作序列取决于提交的指令是预测正确的转移指令、store 指令还是其他指令 (正常提交)。正常提交适用于指令到达 ROB 头部且指令的结果出现在缓存中时的情况; 这时, 处理器将使用该结果更新寄存器, 并移除 ROB 中的指令。提交 store 指令与正常提交类似, 不同之处在于提交 store 指令时更新的是存储器而不是结果寄存器。如果有一条预测错误的转移指令到达 ROB 头部, 则表明推测是错误的。这时, ROB 将被清空, 并从该转移正确的后继指令开始重新执行。如果转移预测正确, 则转移执行完成。

当指令提交后, 指令在 ROB 中的入口将被收回, 寄存器或存储器将被更新, 无须再占用 ROB 的入口。如果 ROB 已被填满, 则只需停止指令的发射, 直至有可用的入口出现为止。现在, 让我们来看看 Tomasulo 算法中的例子是如何采用这种方法的。

例题 假设浮点功能单元的时延与前面的例子相同, 即加法 2 个时钟周期、乘法 6 个时钟周期、除法 12 个时钟周期。对于下面的代码 (即图 2.11 对应的代码), 写出当 MUL.D 准备提交时的状态表。

```
L.D      F6, 32(R2)
L.D      F2, 44(R3)
MUL.D    F0, F2, F4
SUB.D    F8, F6, F2
DIV.D    F10, F0, F6
ADD.D    F6, F8, F2
```

解答: 结果如图 2.15 所示。请注意, 虽然 SUB.D 的执行已经完成, 但是它直至 MUL.D 提交之后才可以被提交。保留站和寄存器状态字段所包含的信息同 Tomasulo 算法中一样 (见 67 页对这些字段的描述)。不同之处在于, 这里用 ROB 入口序号替代了 Qj 和 Qk 字段以及寄存器状态字段中的保留站序号。Dest 字段指向 ROB 入口, 这个入口是该保留站所得到结果的目标地址。

相比, 主要
1. 通过拓宽
发射的情况

的插槽, 则
的控制输入
结果被置于
在出现可用

所需的操
算。在这
指令在这

送), 并且
态。Store 指
字段。否则

上面这个例子阐释了推测执行处理器与动态调度处理器的重要不同。图 2.11 所示为相同的代码在执行 Tomasulo 算法的处理器上的运行情况。比较图 2.11 与图 2.15, 一个重要的区别在于, 在上面的例子中, 最早的指令 (上例中为 MUL.D) 完成之前, 任何指令都不允许执行。而在图 2.11 中, 在 MUL.D 完成之前, SUB.D 和 ADD.D 都已经执行完毕。

这个不同意味着有 ROB 的处理器可以在动态执行代码的同时, 保持精确的中断模型。例如, 如果 MUL.D 指令引起一个中断, 则只需要在该指令到达 ROB 的头部时产生该中断, 并清空在 ROB 中等待的其他指令即可。由于指令是按序提交的, 因此可以精确地实现异常。

一些用户和系统结构设计者认为, 在高性能处理器中, 不精确的浮点异常是可接受的, 因为一旦发生异常, 程序就有可能终止运行; 参见附录 G 中关于这一主题的更多讨论。而对于其他类型的异常 (如页面失效等), 由于程序要在处理完这类异常后继续执行, 因此它们很难容忍这种不精确的情况。

指令按序提交的 ROB 能够在支持推测执行的同时, 保证精确的异常, 如下例所示。

重排序缓存						
入口	忙	指令	状态	目标	值	
1	no	L.D F6,32(R2)	提交	F6	Mem[34 + Regs[R2]]	
2	no	L.D F2,44(R3)	提交	F2	Mem[45 + Regs[R3]]	
3	yes	MUL.D F0,F2,F4	写结果	F0	#2 × Regs[F4]	
4	yes	SUB.D F8,F2,F6	写结果	F8	#2 - #1	
5	yes	DIV.D F10,F0,F6	执行	F10		
6	yes	ADD.D F6,F8,F2	写结果	F6	#4 + #2	

保留站							
名字	忙	Op	Vj	Vk	Qj	Qk	Dest A
Load1	no						
Load2	no						
Add1	no						
Add2	no						
Add3	no						
Mult1	no	MUL.D	Mem[45 + Regs[R3]]	Regs[F4]			#3
Mult2	yes	DIV.D		Mem[34 + Regs[R2]]	#3		#5

浮点寄存器状态										
字段	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder #	3						6		4	5
Busy	yes	no	no	no	no	no	yes	...	yes	yes

图 2.15 当 MUL.D 准备提交时, 尽管有一些其他的指令已经执行完成, 但是只有两条 L.D 指令已经被提交。MUL.D 位于 ROB 的头部, 在这里标出两条 L.D 指令的作用仅仅是帮助理解。尽管 SUB.D 和 ADD.D 指令的结果已经就绪, 且可被其他指令作为源使用, 但是它们只有在 MUL.D 指令提交之后才可以被提交。DIV.D 指令正在被执行, 只是由于它的时延长于 MUL.D, 因此还未执行完成。值字段表示正在保存的值; #X 指向 ROB 入口 X 的值字段。重排序缓存 1 和 2 实际上已经完成了, 但是为了有助于理解仍然将它们在图中表示出来。我们没有标出 load-store 队列的入口, 但是这些入口将保持顺序排列。

例题 考虑下面的代码, 这段代码在介绍 Tomasulo 算法时使用过, 图 2.13 所示为它的执行情况:

```

Loop:  L.D      F0,0(R1)
        MUL.D   F4,F0,F2
        S.D     F4,0(R1)
        DADDIU  R1,R1,#-8
        BNE     R1,R2,Loop    ; 如果 R1 ≠ R2 则转移

```

假设循环中的所有指令已经发射了两次, 同时假设第一个循环体中的 L.D 和 MUL.D 已经提交, 并且其他指令已经执行完成。正常情况下, store 指令将在 ROB 中等待直至所需的有效地址操作数 (即本例中的 R1) 和值 (即本例中的 F4) 就绪。由于我们只考虑浮点流水线, 因此假设 store 所需的有效地址在指令发射后已经计算出来。

解答：结果如图 2.16 的两个表所示。

重排序缓存						
入口	忙	指令	状态	目标	值	
1	no	L.D F0,0(R1)	提交	F0	Mem[0 + Regs[R1]]	
2	no	MUL.D F4,F0,F2	提交	F4	#1 × Regs[F2]	
3	yes	S.D F4,0(R1)	写结果	0 + Regs[R1]	#2	
4	yes	DADDIU R1,R1,#-8	写结果	R1	Regs[R1] - 8	
5	yes	BNE R1,R2,Loop	写结果			
6	yes	L.D F0,0(R1)	写结果	F0	Mem[#4]	
7	yes	MUL.D F4,F0,F2	写结果	F4	#6 × Regs[F2]	
8	yes	S.D F4,0(R1)	写结果	0 + #4	#7	
9	yes	DADDIU R1,R1,#-8	写结果	R1	#4 - 8	
10	yes	BNE R1,R2,Loop	写结果			

浮点寄存器状态									
字段	F0	F1	F2	F3	F4	F5	F6	F7	F8
Reorder #	6				7				
Busy	yes	no	no	no	yes	no	no	...	no

图 2.16 尽管其他指令都已经执行完成，但是只有 L.D 和 MUL.D 指令被提交。这时没有保留站处于工作状态，因此未将它们表示出来。剩余的指令将被尽快提交。前两个重排序缓存是空闲的，但为了完整仍然将它们表示出来

由于在指令提交之前不会实际更新寄存器和存储器，因此当发现转移预测错误时，处理器可以很容易地撤销推测执行的动作。比如在图 2.16 中，假设转移 BNE 在第一次循环中不被执行。则当转移之前的指令到达 ROB 头部时，只需将它们提交即可；而当转移指令到达缓存的头部时，处理器只需要将缓存清空，并从另一路径开始取指令。

在实际执行过程中，当发现转移预测错误后，推测执行的处理器会尽早执行恢复的工作。为了实现恢复，可以将转移预测错误后出现的所有 ROB 入口清空，使 ROB 中转移之前的指令继续执行，同时开始从正确的转移后继取指令。在推测执行处理器中，由于预测错误产生的影响会更严重，性能对转移预测也更加敏感。因此与转移处理有关的各个因素——预测准确率、预测错误的检测时延以及预测错误的恢复时间——也变得更为重要。

只有在异常准备提交时，异常才会被识别并被处理。如果一条推测指令产生了一个异常，则该异常将在 ROB 中重新排序。如果转移预测错误，且执行了不应执行的指令，则异常将随指令一起在清空 ROB 时被清除。如果指令到达了 ROB 的头部，则可以肯定它不再是推测的指令，因此该异常将被允许发生。我们也可以尝试在异常出现且前序转移指令都已处理的情况下，立即处理异常，但异常的情况比转移预测失败更难处理，而且异常发生的概率也比转移预测失败小，所以重要性相对较低。

图 2.17 所示为指令的执行步骤，以及每个步骤所需满足的条件和需要执行的动作。我们给出了转移预测错误以及该错误在指令提交时被处理的情况。尽管推测技术只是对动态调度技术的简单

补充,但是通过对比图 2.17 与 Tomasulo 算法的图 2.12,可以看出,推测技术显著地增加了控制复杂性。此外,值得注意的是,转移预测错误在某种程度上也变得更加复杂了。

状态	需满足的条件	动作或记录工作
发射所有指令	保留站(r)和 ROB(b)均可用	<pre> if (RegisterStat[rs].Busy) /* in-flight instr. writes rs */ { h ← RegisterStat[rs].Reorder; if (ROB[h].Ready) /* Instr completed already */ {RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;} else {RS[r].Qj ← h;} /* wait for instruction */ } else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;} RS[r].Busy ← yes; RS[r].Dest ← b; ROB[b].Instruction ← opcode; ROB[b].Dest ← rd; ROB[b].Ready ← no; </pre>
浮点操作和 store		<pre> if (RegisterStat[rt].Busy) /* in-flight instr writes rt */ { h ← RegisterStat[rt].Reorder; if (ROB[h].Ready) /* Instr completed already */ {RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;} else {RS[r].Qk ← h;} /* wait for instruction */ } else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;} </pre>
浮点操作		<pre> RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd; </pre>
store		<pre> RS[r].A ← imm; </pre>
执行浮点操作	(RS[r].Qj == 0) 和 (RS[r].Qk == 0)	计算结果——操作数在 Vj 和 Vk 中
load 步骤 1	(RS[r].Qj == 0) 和 且队列中没有较早的 store 指令	<pre> RS[r].A ← RS[r].Vj + RS[r].A; </pre>
load 步骤 2	Load 第一步完成, 且 ROB 中所有较早的 store 指令有不同的地址	从 Mem[RS[r].A]中读
store	(RS[r].Qj == 0) 和 且 store 到达队列头部	<pre> ROB[h].Address ← RS[r].Vj + RS[r].A; </pre>
写回所有非 store 的结果	r 中的指令执行完成且 CDB 可用	<pre> b ← RS[r].Dest; RS[r].Busy ← no; ∀x (if (RS[x].Qj == b) {RS[x].Vj ← result; RS[x].Qj ← 0}); ∀x (if (RS[x].Qk == b) {RS[x].Vk ← result; RS[x].Qk ← 0}); ROB[b].Value ← result; ROB[b].Ready ← yes; </pre>
store	r 中的指令执行完成 且 (RS[r].Qk == 0)	<pre> ROB[h].Value ← RS[r].Vk; </pre>
提交	指令在 ROB 头部 (入口 h) 且 ROB[h].ready == yes	<pre> d ← ROB[h].Dest; /* register dest, if exists */ if (ROB[h].Instruction == Branch) { if (branch is mispredicted) {clear ROB[h], RegisterStat; fetch branch dest;} else if (ROB[h].Instruction == Store) {Mem[ROB[h].Destination] ← ROB[h].Value;} else /* put the result in the register destination */ {Regs[d] ← ROB[h].Value;} ROB[h].Busy ← no; /* free up ROB entry */ /* free up dest register if no one else writing it */ if (RegisterStat[d].Reorder == h) {RegisterStat[d].Busy ← no;} </pre>

图 2.17 算法的步骤以及每步所需的前提条件。对将要发射的指令, rd 是目标, rs 和 rt 是源, r 是分配的保留站, b 是分配的 ROB 入口, h 是 ROB 入口的头部。RS 是保留站数据结构。有保留站返回的值称为结果。Registerstat 是寄存器数据结构, Regs 是实际的寄存器, ROB 是重排序缓存数据结构

了控制复

在处理 store 指令时,推测处理器使用的方法与 Tomasulo 算法有一个重要的不同。在 Tomasulo 算法中,store 指令在到达写结果阶段(该阶段保证了有效地址已经被计算完成)且要保存的数据值已经就绪时就可以更新存储器。而在推测处理器中,store 指令只有在到达 ROB 的头部后才可以更新存储器。推测处理器采用的这种方法可以保证存储器在确定指令推测正确后才会被更新。

ly ← no;

图 2.17 对 store 指令做了简化,这种简化在实际执行过程中却不常使用。图 2.17 要求 store 指令在写结果阶段等待寄存器源操作数,该操作数的值即为 store 指令要保存的值;之后,将把这个值从 store 指令保留站的 Vk 字段移动到 ROB 入口的值字段。而在实际过程中,要保存的值只需在 store 指令提交之前到达即可,并且可以通过源指令将这个值直接交给 ROB 入口。这是通过硬件跟踪操作数在 ROB 入口中的就绪时间以及在每条指令完成时搜索 ROB 来寻找相关 store 指令来实现的。

这个补充并不复杂,但是会带来两个影响:我们需要在 ROB 中增加一个字段,图 2.17 已经使用了小号字,而再增加一个字段甚至会使图 2.17 变得更长!尽管图 2.17 做了简化,但是在这个例子中,我们将允许 store 指令跃过写结果阶段,它只需在准备提交前得到要保存的值即可。

同 Tomasulo 算法一样,我们必须在存储器中避免冒险。存储器中的 WAW 和 WAR 冒险是通过推测消除的,这是由于存储器的更新是在 store 指令到达 ROB 头部时按序发生的,这时肯定不会仍有在等待的前序 load 或 store 指令存在。由于以下两种原因,存储器中的 RAW 冒险仍将存在:

1. 如果某个 store 指令占用的 ROB 入口是活跃的,且它的目标字段与 load 指令的 A 字段匹配,则不允许 load 指令进入执行的第二阶段。
2. 为计算 load 指令的有效地址,维护 load 指令与前序的 store 指令之间的程序顺序。

这两条限制使得在 store 指令向某个存储器地址写数据之前,任何访问该存储器地址的 load 指令都不会被执行。当发生 RAW 冒险时,一些推测处理器会采取将值旁路掉的方法,即由 store 指令将值直接交给 load 指令。另一种方法是通过值预测的方式,预测潜在的冒险;我们将在 2.9 节中讨论这种方法。

尽管上述对推测执行的解释是针对浮点操作的,但是这种技术也可以很容易地扩展到定点寄存器和功能单元,我们将在“综合”一节中讨论这个问题。实际上,由于在定点程序中转移的可预测性较低,因此推测技术更适用于定点程序。此外,通过将这些技术扩展为每个时钟周期内发射和提交多条指令,这些技术可以应用于多发射处理器。实际上,对于这些处理器来说,推测技术可能会具有更强的吸引力。因为在这些处理器中,通过编译器的支持,一些实用的技术可能会在基本块之间获得足够的指令级并行度。

2.7 采用多发射和静态调度技术开发指令级并行

前几节中介绍的技术可以用来消除数据相关和控制相关引起的停顿,从而获得理想 CPI,即 CPI 为 1。为了进一步提高性能,我们需要将 CPI 减小到 1 以下。但是在每个时钟周期发射一条指令的前提下,我们是无法实现这个目标的。

多发射处理器的目标是允许在一个时钟周期内发射多条指令。我们将在后面几节中讨论多发射处理器。当前主要有以下三种风格的多发射处理器:

1. 静态超标量调度处理器。
2. VLIW (超长指令字) 处理器。
3. 动态调度超标量处理器。

以上两种超标量处理器均为每时钟周期发射多条指令，静态调度的处理器使用按序执行，动态调度的处理器使用乱序执行。

与超标量处理器不同，VLIW处理器每时钟周期发射固定数目的指令，这些指令或者被组织成一条长指令的形式，或者被组织成一个固定的指令包，指令间的并行度由指令显式地表示出来。VLIW处理器采用编译器静态调度的方式。Intel和HP在创建IA-64系统结构时（在附录G中介绍），也将这种系统结构风格命名为EPIC，即显式并行指令计算机。

尽管静态超标量每时钟周期发射多条指令，而不是固定数目的指令，但是它在概念上却与VLIW十分接近，因为这两种方法都依赖编译器为处理器调度代码。由于发射宽度的增长会削弱静态调度超标量的优势，因此静态超标量主要应用于发射宽度有限（一般情况下只有两条指令）的情况。而对于更大的宽度，大多数设计者会选择VLIW或动态调度超标量来实现。由于VLIW和静态调度超标量处理器在硬件及编译技术方面是相似的，因此我们将在这一节中集中讨论VLIW。结合本节的知识，读者可以自学有关静态调度超标量处理器的内容。

图2.18概括了多发射的基本方法，描述了它们的主要技术特征，并给出了使用各种技术的处理器。

名称	发射结构	冒险检测	调度	主要特征	实例
超标量（静态）	动态	硬件	静态	乱序执行	主要在嵌入式领域： MIPS和ARM
超标量（动态）	动态	硬件	动态	部分乱序执行， 但是不带推测	目前还没有
超标量（推测）	动态	硬件	带有推测	带推测的乱序 执行	Pentium 4, MIPS R12K, IBM Power 5
VLIW/LIW	静态	软件为主	静态	所有冒险由编译 器裁定并标识 （通常是隐式的）	大多数应用在嵌入式领 域，如TI C6x
EPIC	静态为主	软件位主	大多数是 静态	所有冒险由编译器 显示裁定并标示	Itanium

图2.18 多发射处理器中应用的5种主要方法以及它们的主要特征。这一章主要关注硬件技术，它们被用在所有的超标量处理器中。附录G集中介绍了基于编译器的方法。在IA-64系统结构中使用的EPIC方法扩展了早期VLIW方法的许多概念，实现了静态方法和动态方法的组合

基本的VLIW方法

VLIW采用多个独立的功能单元。与尝试向单元中发射多条独立指令的方法不同，VLIW将多个运算打包成一条长指令，或者要求发射包中的指令满足相同的约束。由于这两种方法并没有本质的不同，我们不妨假设多个运算组成一条长指令，最初的VLIW方法就是这样设计的。

当最大发射率上升时，VLIW的优势会更加明显，因此我们将集中讨论宽发射处理器的情况。实际上，对于简单的双发射处理器来说，超标量的开销有可能是最小的。许多设计人员相信一个四发射处理器的开销是可控的，而我们将在下一章看到，开销的增长是限制宽发射处理器的一个首要因素。

让我们考虑一条包含5个运算的指令在VLIW处理器上的运行情况。该指令包括一个定点运算（也可能是一个转移）、两个浮点运算以及两个存储器运算，指令应当为每个功能单元对应一组

字段——也许每个单元 16~24 bit, 从而使指令的长度大约在 80~120 bit 之间。而作为对比, Intel Itanium 1 和 2 的每个指令包只含有 6 个运算。

为了使功能单元始终处于工作状态, 代码序列必须含有足够的并行度, 以填满功能单元的可用运算槽。代码序列中的并行度是通过将循环展开, 在每个单独的、更大的循环体中进行代码调度而显现的。如果通过展开得到的是无转移代码, 则可以使用局部调度技术, 针对每个单独的基本块进行调度。如果并行度的开发要求跨转移调度代码, 则需要使用更加复杂的全局调度算法。全局调度算法不仅在结构上更加复杂, 而且由于跨转移移动代码的代价是非常昂贵的, 因此全局调度算法还必须在优化过程中综合各种因素做出权衡, 这进一步增加了它的复杂性。

在附录 G 中, 我们将讨论踪迹调度, 即一种专门为 VLIW 设计的全局调度技术; 我们还将探讨消除条件转移的特殊硬件支持, 以便扩展局部调度算法的可用性, 提升全局调度算法的性能。

现在, 我们将使用循环展开技术生成无转移的代码序列, 以此为基础, 我们可以使用局部调度技术增强 VLIW 指令, 并集中研究 VLIW 处理器的运行情况。

例题 假设有一个 VLIW 处理器, 它每个时钟周期可发射两个访问存储器操作、两个浮点操作以及一个定点操作或转移操作。写出循环 $x[i] = x[i] + s$ (见 52 页上的 MIPS 代码) 在该处理器上展开的情况。忽略转移延迟, 尽可能地展开循环以消除停顿。

解答: 代码如图 2.19 所示。循环被展开, 形成 7 个循环体副本, 消除所有停顿 (即完全空的发射周期), 运行 9 个时钟周期。这种代码将保持每 9 个时钟周期产生 7 个结果的速率, 或者说每获得 1 个结果花费 1.29 个时钟周期, 这种速度几乎是 2.2 节中介绍的使用循环展开和代码调度的双发射超标量速度的两倍。

存储器引用 1	存储器引用 2	浮点操作 1	浮点操作 2	定点操作 / 转移
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

图 2.19 内循环的 VLIW 指令, 替换展开序列。假设在没有转移延迟的情况下, 这段代码需要花费 9 个时钟周期; 一般情况下转移延迟也需要进行调度。发射率为每 9 个时钟周期发射 23 个操作, 即每时钟周期 2.5 个操作。其效率 (即含有操作的可用插槽所占的百分比) 大约为 60%。达到这个发射率需要大量的寄存器, 其数量远远多于 MIPS 一般情况下所需的寄存器数。上述 VLIW 代码序列至少需要 8 个浮点寄存器, 而同样情况下 MIPS 处理器只需要 2 个, 即使在展开和调度时也只需要 5 个。

最初的 VLIW 模型由于存在技术和逻辑上的问题, 因此性能有限。技术问题主要源于代码量的快速增加以及锁步操作带来的限制。代码量的快速增加来自两方面因素的共同影响。首先, 为了在无转移代码片断中生成足够数量的运算, 必须要尽可能多地展开循环 (如早些时候的例子), 因此会导致代码量的增加。其次, 当指令未被充满时, 在指令译码阶段闲置的功能单元会填充无用的

比特,从而增加代码量。在附录G中,我们将讨论软件调度方法,如软件流水线等,这些方法可以在展开循环的同时尽量限制代码的膨胀。

为了克服代码量的快速增加带来的影响,有时也采用智能译码方法。例如,让所有的功能单元使用一个立即数字段。另一种方法是在主存中压缩指令,当指令被读入Cache或被译码时再将它们解压。在附录G中,我们还会讨论一些其他的技術,同时验证在IA-64中存在的代码膨胀问题。

早期VLIW中的操作是锁步的,并且没有检测冒险的硬件。在这种结构中,由于所有的功能单元必须保持同步,因此任意一个功能单元的停顿都将引起整个处理器的停顿。尽管编译器可以对确定的功能单元进行调度以阻止停顿,但是想要预测哪些数据访问会引起停顿以及对它们进行调度是非常困难的。因此,所需Cache的阻塞会引起所有功能单元的停顿。当发射率和存储器操作的次数都变得越来越大时,这种同步限制将是无法容忍的。在最近的处理器中,功能单元的操作变得越来越独立,编译器在指令发射阶段避免冒险,而在指令发射后,由硬件负责检测和支持指令的异步执行。

二进制代码的兼容性是VLIW面临的一个主要的逻辑问题。在严格的VLIW方法中,代码序列既要符合指令系统的定义,也要考虑具体流水线的结构,包括功能单元及其时延等。因此,根据功能单元数量的不同和单元时延的不同,需要有不同的版本的代码。与超标量设计相比,VLIW的这个限制使得在不同版本的或是拥有不同发射带宽的机器之间移植代码变得更加困难。当然,对于一个新的超标量设计来说,性能的改善可能是以重新编译为代价的。尽管如此,运行老版本二进制代码的能力仍然是超标量方法的一个优势。

以IA-64系统结构为代表的EPIC方法解决了在早期VLIW设计中遇到的许多问题,包括软件推测的扩展以及在保证二进制代码兼容性的同时克服硬件依赖的限制等。

开发指令级并行度是所有多发射处理器所面临的一个共同挑战。有些情况下,向量处理器可以高效地执行浮点程序中的一些简单循环,而多发射处理器则需要将循环展开才可获得足够的并行度(见附录F中的介绍)。对于这类应用来说,我们很难确定多发射处理器是否真的优于向量处理器。当代价相同时,向量处理器甚至可能会更快。而与向量处理器相比,多发射处理器的优势在于它能够从结构化较差的代码中开发并行度,而且有能力缓存所有格式的数据。正是这些原因使得发射处理器成为开发指令级并行度的首选方法,而向量处理器只是作为多发射处理器的主要补充。

2.8 采用动态调度、多发射和推测方法开发指令级并行

到目前为止,我们已经看到了动态调度机制、多发射机制和推测机制各自是如何工作的。一节中,我们将把三者结合到一起,这将是与现代微处理器十分相似的微系统结构。虽然化起见,我们将只考虑发射速率为每时钟周期两条指令的情况,但是本节中介绍的概念和每时钟周期发射三条或三条以上指令的现代处理器是一致的。

现在让我们来对Tomasulo算法进行扩展,以使其支持双发射超标量流水线,假设该流水线的点单元和浮点单元是各自独立的,它们每时钟周期可以初始化一个操作。由于会破坏程序的结构,因此我们不能采用乱序机制将指令发射到保留站中去。为了充分发挥动态调度的优势,允许流水线在一个时钟周期内发射任意两条指令的组合,通过调度硬件为运算分配定点和浮点单元。由于定点指令和浮点指令之间的交互十分关键,因此我们还需要扩展Tomasulo算法以使其支持定点和浮点功能单元以及寄存器,同时将其与推测执行技术整合在一起。

为了能够每时钟周期发射多条指令,动态调度处理器通常会采用两种不同的方法,这两者的关键在于保留站的分配和流水线控制表的更新。一种方法是使指令在半个时钟周期内通过

8 = 23

方法可以段,这样就可以在一个时钟周期内处理两条指令。另一种方法是为同时处理两条指令建立必需的逻辑,包括指令间可能存在的相关性等。一个每时钟周期能发射4条或4条以上指令的现代超标量处理器均使用了上述两种方法:都采用流水线方式并拓宽了发射逻辑。

为了将推测的动态调度和多发射结合在一起,我们必须使流水线后台满足一个额外的要求,即每时钟周期都能完成和提交指令。与多发射指令的情况类似,虽然想法很简单,但实现起来也许会和寄存器重命名过程一样复杂。下面让我们通过一个例子来看看这些概念是怎样结合在一起的。

例题 考察下面的循环在有推测和无推测的情况下,在双发射处理器上的执行过程。该循环递增一个定点数组中的所有元素:

```
Loop:  LD    R2,0(R1)    ; R2 = 数组元素
      DADDIU R2,R2,#1    ; 递增 R2
      SD    R2,0(R1)    ; 保存结果
      DADDIU R1,R1,#8    ; 指针加 8
      BNE   R2,R3,LOOP  ; 不是最后一个元素时转移
```

假设有单独的定点功能单元进行有效地址计算、ALU 运算和条件转移计算。分别给出循环的前三个循环体在有推测和无推测处理器中执行时的时间表。

解答: 图 2.20 和图 2.21 分别为上述代码在有推测和无推测的双发射动态调度处理器中的表现。在这个例子中,转移是限制性能的主要因素,推测能够很好地解决这个问题。在推测处理器中,第三个转移在第 13 个时钟周期执行,而在无推测流水线中则在第 19 个时钟周期执行。由于无推测流水线的完成率要远远低于发射率,因此当发射更多的迭代时,无推测流水线将陷入停顿状态。在转移条件确定之前计算 load 指令的有效地址可以改进无推测处理器的性能,但是除非允许推测的存储器访问,否则这种改进将仅限于每次迭代 1 个时钟周期。

迭代	指令	发射时刻	执行时刻	存储器访问时刻	写 CDB 时刻	注释
1	LD R2, 0(R1)	1	2	3	4	第一次发射
1	DADDIU R2, R2, #1	1	5		6	等待 LW
1	SD R2, 0(R1)	2	3	7		等待 DADDIU
1	DADDIU R1, R1, #8	2	3		4	直接执行
1	BNE R2, R3, LOOP	3	7			等待 DADDIU
2	LD R2, 0(R1)	4	8	9	10	等待 BNE
2	DADDIU R2, R2, #1	4	11		12	等待 LW
2	SD R2, 0(R1)	5	9	13		等待 DADDIU
2	DADDIU R1, R1, #8	5	8		9	等待 BNE
2	BNE R2, R3, LOOP	6	13			等待 DADDIU
3	LD R2, 0(R1)	7	14	15	16	等待 BNE
3	DADDIU R2, R2, #1	7	17		18	等待 LW
3	SD R2, 0(R1)	8	15	19		等待 DADDIU
3	DADDIU R1, R1, #8	8	14		15	等待 BNE
3	BNE R2, R3, LOOP	9	19			等待 DADDIU

图 2.20 在无推测的双发射流水线中,发射、执行和写结果的时刻表。请注意, BNE 后的 LD 指令不能提早执行,它必须等待转移结果的确定。在这类程序中,数据相关的转移不能被提前解决,推测技术会在这类程序中发挥作用。由于有分开的功能单元用来进行地址计算、ALU 运算和条件转移计算,所以多条指令可以在一个时钟周期内执行。图 2.11 所示为同样的例子在有推测的处理器中的运行情况

循环 体号	指令	发射 时刻	执行 时刻	存储器访 问时刻	写 CDB 时刻	提交 时刻	注释
1	LD R2, 0(R1)	1	2	3	4	5	第一次发射
1	DADDIU R2, R2, #1	1	3		6	7	等待 LW
1	SD R2, 0(R1)	2	3			7	等待 DADDIU
1	DADDIU R1, R1, #8	2	3		4	8	按序提交
1	BNE R2, R3, LOOP	3	7			8	等待 DADDIU
2	LD R2, 0(R1)	4	5	6	7	9	没有执行延迟
2	DADDIU R2, R2, #1	4	8		9	10	等待 LW
2	SD R2, 0(R1)	5	6		7	10	等待 DADDIU
2	DADDIU R1, R1, #8	5	6			11	按序提交
2	BNE R2, R3, LOOP	6	10			11	等待 DADDIU
3	LD R2, 0(R1)	7	8	9	10	12	尽可能提前
3	DADDIU R2, R2, #1	7	11		12	13	等待 LW
3	SD R2, 0(R1)	8	9		10	13	等待 DADDIU
3	DADDIU R1, R1, #8	8	9			14	提前执行
3	BNE R2, R3, LOOP	9	13			14	等待 DADDIU

图 2.21 在有推测的双发射流水线中, 发射、执行和写结果的时刻表。请注意, 由于是推测的, 因此 BNE 后面的 LD 指令可以提早执行

上面这个例子清楚地表明, 在存在数据相关转移的程序中, 推测方法具有很大的优势, 而如果不使用推测方法, 则性能将会受到限制。但是, 推测起到的作用要取决于转移预测的准确率。不准确的转移预测不但不会改进性能, 而且会对性能产生不利影响。

2.9 指令传送和推测的高级技术

在高性能流水线特别是多发射流水线中, 仅靠准确地预测转移是不够的, 还需要能够传送高带宽的指令流。在现代多发射处理器中, 这意味着每时钟周期要传送 4~8 条指令。我们将首先研究高指令传送带宽的方法。之后将讨论一组与实现高级推测技术有关的问题, 包括寄存器重命名和排序缓存的比较、深度推测以及能够进一步提高指令级并行度的值预测技术。

提高取指令带宽

多发射处理器要求每时钟周期取到的平均指令数目至少不低于平均吞吐量。当然, 这要求指令 Cache 的路径足够宽, 但最困难的问题还是转移的处理。这一节中会研究两种处理转移的方法。之后我们将讨论现代处理器是如何将指令预测和预取功能整合在一起的。

转移目标缓存

为了减小简单五段流水线和深度流水线的转移代价, 必须确定当前正在译码的指令是否是转移指令, 如果是, 那么下一条指令的地址又是什么。如果指令是转移指令并且已经知道下一条指令的地址, 那么我们就可以将转移代价降为 0。我们把为转移的后继指令保存预测地址的转移预测称为转移目标缓存或转移目标 Cache。

由于转移目标缓存要预测下一条指令的地址, 并且在指令译码结束前将预测地址发送出去, 此我们必须确定取到的指令是否是一条被预测为将被选中的转移指令。如果所取指令的地址同缓存中的一条指令地址匹配, 则相应的预测指令地址将被作为下一条指令地址。转移预测缓存结构在本质上与 Cache 的硬件结构是一致的。

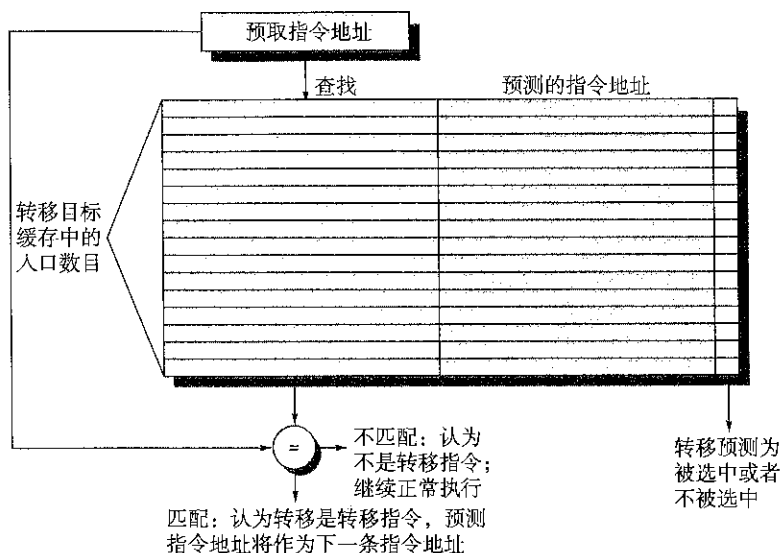


图 2.22 转移目标缓存。将预取指令地址同保存在第一列中的一组指令地址进行匹配；这些表示已知的转移地址。如果指令地址与这些入口中的某一个匹配，则认为预取指令是一条将要被选中的转移指令；而第二个字段，即预测指令地址字段，包含对转移后继指令的指令地址的预测。取指令将立刻从这个地址开始。第三个字段是可选的，可以用做额外的预测状态位

如果在转移目标缓存中发现了一个匹配的入口，则取指令将立即从预测指令地址开始。值得注意的是，与转移预测缓存不同，由于要在确定指令是否是转移之前将预测地址发送出去，因此转移目标缓存的预测入口必须与指令完全匹配。如果处理器不对这种匹配进行核实，那么在当前指令不是转移的情况下，发送出去的预测地址就将是错误的，这会降低处理器的速度。在转移目标缓存中，我们只需要保存那些预测为被选中的转移，而对于不被选中的转移，只需要按线性顺序取下一条指令即可，这和非转移指令的情况相同。

图 2.23 所示为在简单五段流水线中使用转移目标缓存的详细步骤。从这幅图我们可以看出，在缓存中存在匹配的转移预测入口且预测正确的情况下，是不存在转移延迟的。否则，将至少付出 2 个时钟周期的转移代价。处理缓存缺失和预测错误是一个艰巨的挑战，因为我们必须在重写缓存入口的同时停止取指令的操作。所以，要想将转移代价减少到最小，我们必须加速这个处理过程。

为了评估转移目标缓存的工作表现，必须首先确定在各种可能情况下所付出的代价。图 2.24 针对简单五段流水线给出了这方面的信息。

例题 假设单个预测错误的代价周期如图 2.24 所示，确定一个转移目标缓存总的转移代价。假设预测准确率和缓存命中率如下：

- 预测准确率为 90%（对于缓存中的指令）。
- 缓存命中率为 80%（对于预测为被选中的转移）。

解答：我们通过调查下面这两种情况的可能性计算代价：预测转移被选中，但最终没有被选中；转移被选中，但该转移不在缓存中。这两种情况的代价均为 2 个时钟周期。

$$\begin{aligned}\text{概率(转移在缓存中, 但不被选中)} &= \text{缓存命中率} \times \text{预测错误率} \\ &= 90\% \times 10\% = 0.09\end{aligned}$$

$$\text{概率(转移不在缓存中, 且不被选中)} = 10\%$$

$$\text{转移代价} = (0.09 + 0.10) \times 2$$

$$\text{转移代价} = 0.38$$

这个代价低于我们在附录A中评估的转移时延所造成的转移代价(大约为0.5个时钟周期)。当流水线长度和转移延迟增加时, 通过动态转移预测得到的性能改善也会随之增长; 此外, 使用更准确的预测器也会获得更大的性能优势。

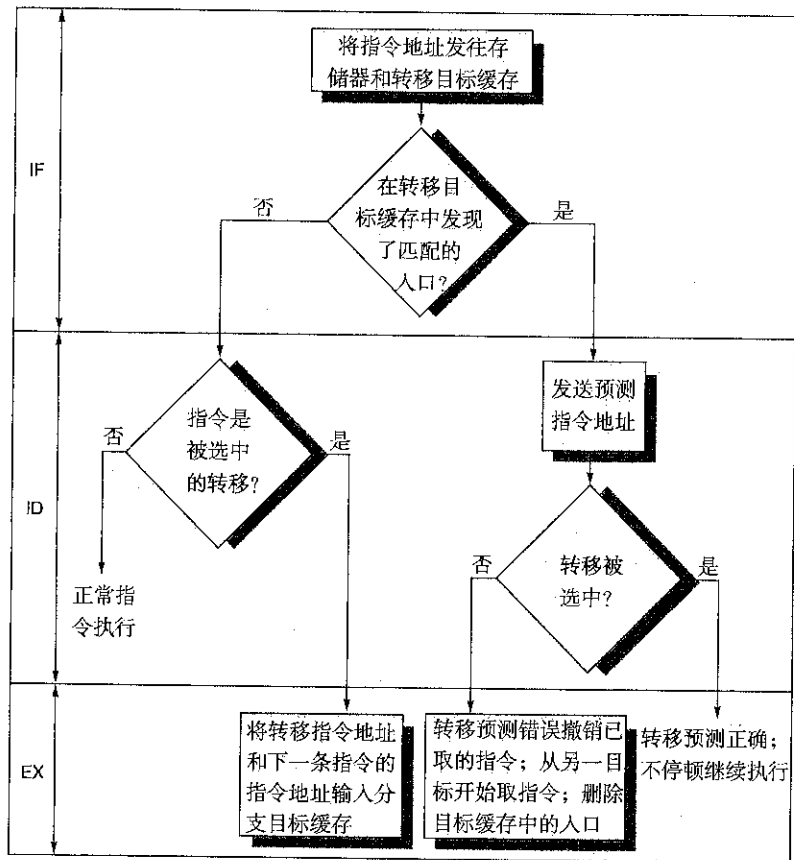


图 2.23 转移目标缓存处理指令所涉及的步骤

缓存中的指令	预测	实际转移动作	代价周期
yes	选中	选中	0
yes	选中	不被选中	2
no		选中	2
no		不被选中	0

图 2.24 转移是否在缓存中以及转移实际表现的各种可能组合的代价, 假设我们只在缓存中保存被选中的转移。在所有预测都正确且转移在目标缓存中的情况下, 转移代价为0。如果转移预测错误, 则其代价为2个时钟周期, 即用正确信息更新缓存的1个时钟周期(在这段时间内取指令操作将被停止)加上从正确的转移之后继续重取指令的1个周期。如果转移不在目标缓存中, 且转移被选中, 则其代价为2个时钟周期, 在这段时间里缓存将被更新

转移目标缓存的另外一种形式是在缓存中保存一条或多条目标指令,以此作为预测目标地址的替代或补充。这种形式有两个潜在的优势。首先,这种方法允许转移目标缓存的访问时间超过两个相继取指令操作的时间间隔,也可能允许更大的转移目标缓存。其次,通过缓存实际的指令可以实现优化,我们称之为**转移隐含**(branch folding)。使用转移隐含,我们可以将无条件转移以及某些情况下条件转移的转移代价降为0时钟周期。现在假设一个转移目标缓存已经保存了预测路径上的指令,让我们来考虑通过无条件转移地址访问该缓存的情况。无条件转移的唯一作用就是改变指令地址。因此,当转移目标缓存被命中并且表明当前的转移地址是无条件转移时,流水线只需用转移目标缓存中的指令替换掉从Cache中返回的指令(这里是无条件转移)即可。如果处理器每时钟周期能够发射多条指令,那么缓存需要提供多条指令以获得最大性能收益。在某些情况下,当条件代码被重置时,转移隐含可能会消除条件转移的代价。

返回地址预测器

当尝试提高推测的机会和准确性时,我们将面临预测间接转移所带来的挑战。间接转移的目标地址是在运行时决定的。尽管高级编程语言也会为间接程序调用、case选择语句以及FORTRAN中的goto指令生成这类转移,但是间接转移的大部分还是来自于过程返回。例如,在SPEC95基准测试程序中,过程返回在所有转移中所占的比例平均大约为15%,而在间接转移中更是占据绝大部分。而在C++和Java这样的面向对象语言中,过程返回的频率甚至更加频繁。这就是为什么我们要把注意力集中到过程返回上的原因。

尽管过程返回可以通过转移目标缓存来预测,但是在过程被多方调用且调用时间分散的情况下,这种预测技术的准确率会很低。例如,在SPEC CPU95中,对于这类返回转移,深度转移预测器也只能达到不及60%的准确率。为了解决这个问题,一些设计方案使用堆栈作为返回地址缓存。该结构用来缓存最近的几个返回地址:当调用发生时将返回地址压入堆栈,调用返回时再将其弹出。如果Cache足够大的话(即相当于最大的调用深度),它将出色地预测返回地址。图2.25所示为由0~16个元素组成的返回缓存在SPEC CPU95基准测试程序中的性能情况。在3.2节中学习指令级并行时,我们将使用类似的返回预测器。

集成的取指令单元

为了满足多发射处理器的要求,许多设计师选择了实现集成取指令单元的方法,即将取指令作为一个单独且自主的单元来实现,由该单元为流水线的其他部分提供指令。实际上,这等价于考虑到多发射的复杂性,不再将取指令视为一个有效的单独流水段。

在最新的设计中,集成取指令单元主要整合了以下几种功能:

1. **集成的转移预测**: 转移预测器成为了取指令单元的一部分,并且持续对转移进行预测,以驱动流水线的取指令操作。
2. **指令预取**: 为实现在一个时钟周期内传送多条指令,取指令单元可能需要提前取指令。取指令单元自主管理指令的预取(见第5章对这类技术的讨论),并将它同转移预测整合在一起。
3. **指令存储器访问和缓存**: 一个时钟周期内取多条指令会引起一系列的复杂问题,比如取多条指令要求访问多路Cache的问题。取指令单元屏蔽了这种复杂性,它使用预取法避免跨Cache块引起的代价。取指令单元也提供缓存功能,作为一个按需单元为发射流水段提供所需的指令。

设计师们的努力使得每时钟周期可执行的指令数目不断增长,在这种情况下,取指令将成为一个越来越明显的瓶颈,我们需要采用一些新的、更好的方法来提高指令的传送速率。在 Pentium 4 中使用的 Cache 跟踪就是一种典型的方法,我们将在附录 C 中讨论这种方法。

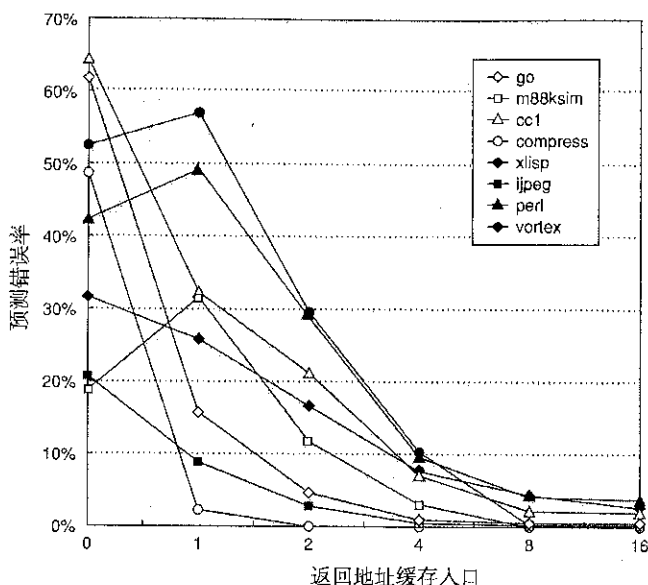


图 2.25 在 SPEC CPU95 基准测试程序中,以堆栈方式操作的返回地址缓存的准确率。准确率即预测准确的返回地址所占的比例。0 个入口的缓存相当于一个标准的转移预测缓存。由于一般情况下调用深度都不大,因此较小的缓存也可以取得较好的工作效果。以上数据来源于 Skadron 等(1999),采用了一种改进的方法防止 Cache 返回地址的误用

推测：实现问题和扩展

我们将在这一节中探讨涉及推测实现的三个问题,从使用寄存器重命名开始,这种方法几乎完全替代了重排序缓存。之后,我们将讨论一种重要的扩展方法来推测控制流,这种方法称为值推测。

支持推测：寄存器重命名与重排序缓存

重排序缓存(ROB)的一种替代方法是结合使用大量寄存器和寄存器重命名技术。这种方法以 Tomasulo 算法中使用的重命名思想为基础,并对它做了扩展。在 Tomasulo 算法中,执行过程中的任一时刻,结构可见寄存器(R0, ..., R31 和 F0, ..., F31)的值都被保存在寄存器集与保留站的组合中。补充了推测技术后,寄存器的值也可以临时保存在 ROB 中。不论是哪一种情况,如果处理器长时间不发射指令,则所有指令都将被提交,而寄存器的值将出现在与结构可见寄存器直接对应的寄存器文件中。

在寄存器重命名方法中,物理寄存器的扩展集被用来保存结构可见寄存器和临时值。即扩展寄存器替代了 ROB 和保留站的功能。在发射流水段期间,重命名过程将系统结构寄存器的名字映射为扩展寄存器集中物理寄存器的序号,为目标分配一个空闲寄存器。WAW 和 WAR 冒险通过重命名目标寄存器来消除,而由于保存目标指令的物理寄存器在指令提交之前不会成为系统结构寄存器,因此,推测恢复的问题也得到了解决。重命名映射是一个简单的数据结构,它提供了当前与某个特定系统结构寄存器对应的物理寄存器序号。重命名映射在结构上和功能上都与 Tomasulo 算法中的

寄存器状态表相似。当指令提交时,重命名表将被永久更新为与实际的系统结构寄存器对应的物理寄存器,从而完成对处理器状态的更新。

与ROB方法相比,重命名方法的一个优势在于指令提交的简化,这只需要两个简单的动作:将系统结构寄存器序号与物理寄存器序号之间的映射标记为不再推测,释放所有保存系统结构寄存器旧值的物理寄存器。在保留站的设计中,在指令完成执行之后,它使用的保留站将被释放;而ROB的入口则是在相应指令提交以后被释放的。

在寄存器重命名方法中,撤销分配寄存器的工作会更加复杂,这是由于在释放物理寄存器之前,我们必须确定它与任何系统结构寄存器不再对应,且对该寄存器的使用已全部完成。而在系统结构寄存器被重写、使重命名表指向别处之前,物理寄存器将始终保持与系统结构寄存器的对应关系。这就是说,如果没有重命名入口指向特定的物理寄存器,则表明它已经不再对应于系统结构寄存器了。但是对物理寄存器的使用可能仍未完成。通过检查指令队列中所有指令的源寄存器清单,处理器可以判定当前的情况。如果在源寄存器清单中没有找到给定的物理寄存器,且它没有被指定为系统结构寄存器,那么该物理寄存器将被收回并重新分配。

另一种可供选择的方法是,处理器只需等待对相同的系统结构处理器执行写操作的另一条指令提交。在这个时刻,我们可以确定对旧值的使用已经全部完成。尽管这种方法会在绑定物理寄存器上花费一些不必要的时间,但是它容易实现,并且已经在最新的一些超标量处理器中得到了应用。

如果系统结构寄存器总是在不断地改变,那么我们怎样才能知道哪个寄存器是系统结构寄存器呢?虽然这在程序执行的大多数时间里不是问题,但是在有些情况下,其他的进程(比如操作系统)必须要准确地知道某个特定系统结构寄存器的内容保存在何处。为了搞清楚这个问题,假设处理器在一段时间内没有发射指令。最终,流水线中的所有指令都将被提交,而系统结构可见寄存器同物理寄存器之间的映射也将稳定下来。在这个时刻,系统结构可见寄存器是物理寄存器的一个子集,而物理寄存器中与系统结构寄存器无关的值都是不需要的。这样一来,我们就可以轻松地将系统结构寄存器的值移动到物理寄存器的一个固定子集中,从而使它们可以和其他进程自由通信。

在过去几年中,大多数高端处理器,包括Pentium系列、MIPS R12000以及Power和PowerPC等,都不约而同地采用了寄存器重命名方法,扩充了20~80个寄存器。在提交之前,所有的结果都被分配在虚拟寄存器中,这些额外寄存器替代了ROB的主要功能,并且在很大程度上决定了某个时刻(在发射和提交之间)可以执行的指令的数量。

推测的代价

推测技术的明显优势在于它能够尽早发现可能导致流水线停顿的事件,比如Cache缺失等。然而这种潜在的优势却伴随着明显的弱点。推测不是免费的:它需要消耗时间、消耗能量,恢复错误的推测还会降低流水线的性能。此外,为了支持更高的指令执行速率,处理器必须需要额外的资源。最后,如果推测会引起异常的发生,比如Cache或TLB的缺失,而在无推测的情况下本不应出现这些事件,那么在这种情况下潜在的性能损失将更加严重。

为了最大限度地保持推测的优势,并将推测的代价限制在最小,大多数推测处理器只允许低代价的异常事件发生(比如一级Cache缺失)。如果发生了代价昂贵的异常事件,如Cache缺失或TLB缺失,则处理器在处理该事件之前,必须等待直至引起该事件的指令推测性质消失。尽管这会轻微影响一些程序的性能,但是对于其他情况,特别是在频繁发生此类事件且转移预测率不够理想的情况下,这种方法能够有效地避免性能损失。

在 20 世纪 90 年代, 推测技术的下降趋势还不那么明显。但是随着处理器的发展, 为推测术所付出的代价以及宽发射和推测所受的限制都变得越来越明显。我们将在下一章继续讨论这些问题。

多转移推测

在此前的例子中, 在必须对另一个转移做出推测之前, 可以先处理一个转移。有三种情况可从同时推测多个转移中受益: 转移频率非常高, 转移明显成簇, 以及功能单元延迟较长。在前两种情况中, 获得高性能意味着同时有多条转移被推测, 甚至是每时钟周期处理一条以上的转移。数据库程序以及其他一些结构化较差的定点计算通常具有此类特性, 因此, 推测多条转移对于此类程序格外重要。同样, 多转移作为一种在长流水线中避免停顿的方法, 长延迟的功能单元也会使它的重要性得到提升。

在多转移推测中, 推测错误的恢复要略微复杂一些, 但是也不难实现。更加复杂的技术是每时钟周期预测并推测多条转移。IBM Power2 每时钟周期可以处理两条转移指令, 但是不能对其他任何的转移指令进行推测。直到 2005 年, 还没有一种处理器能够将推测与每时钟周期处理多条转移完整地结合在一起。

值预测

值预测是一种提高程序中可用指令级并行度的技术。值预测尝试对指令将要产生的结果的值进行预测。很明显, 由于大多数指令在每次执行时产生的结果的值并不相同 (至少是一组值中的不同的值), 因此值预测的成功率是有限的。但是, 值预测对于某些特定类型的指令来说却是可行的, 比如读取一个常量池, 或是一个相对稳定变量的 load 指令。此外, 当指令的值是在一小组可能的值中进行选择时, 也许可以通过不把结果的值与实例关联起来进行预测。

如果能够显著增加可用的指令级并行度, 那么值预测将是很有价值的。而在像 load 这类的代码中, 一个值通常被一连串相关计算使用, 在这种情况下值预测将明显增加代码中的可用指令级并行度。由于值预测被用来增强推测能力, 而错误的推测将对性能产生不利影响, 因此预测准确率十分关键。

大部分关于值预测的研究集中在 load 指令上。我们可以通过检测 load 返回的值与最近几次 load 执行结果的匹配比率来确定值预测可能达到的最高准确率。最简单的情况是检查 load 的返回值与上一次执行时得到的值是否匹配。对于 SPEC CPU2000 基准测试程序来说, 这种匹配的可能性大约在 5%~80% 之间。如果我们将 load 的值与最近 16 次的执行结果进行匹配, 则潜在的匹配比例将会升高, 该比率在一些基准测试程序中达到了 80%。当然, 与最近 16 次执行结果中的 1 个结果相匹配并不能告诉我们预测哪个值会比较合理, 但这意味着即使有其他信息的支持, 值预测技术可能达到的预测准确率也不会超过 80%。

鉴于预测错误会付出高昂的代价, 且预测错误率相当高 (20%~50%), 因此, 研究人员开始集中精力对 load 指令的可预测性进行评估, 以确定哪些 load 指令的可预测性更高, 并将值预测的应用范围限定在这类指令范围内。这虽然能够降低预测的错误率, 但是可选择的预测对象也相应减少了。如果我们只尝试预测那些总是返回相同结果的 load 指令, 那么在所有的 load 指令中, 只有大约 10%~15% 是可预测的。针对值预测的研究还在继续。但是现有的结果还不具有足够的说服力, 目前还没有一款应用这项技术的商用处理器出现。

一种与值预测有关且简单实用的方法称为地址别名预测。地址别名预测是一种预测两条 store 指令或一条 store 指令与一条 load 指令是否指向相同地址的简单技术。如果它们指向不同的地址, 则可以安全地交换。否则就必须等待, 直到指令访问的存储器地址确定下来为止。由于不必对地址的

展,为推测实际值进行预测,而只需预测这些地址是否冲突,所以这种预测既简单又稳定。它也因此在一些处继续讨论这处理器中得到了应用。

2.10 综合: Intel Pentium 4

三种情况可
较长。在前两
亡的转移。数
对于此类程
记也会使它的
的技术是每
是不能对其他
明处理多条转

Pentium 4是一款支持带推测多发射的深度流水线。在这一节中,我们将集中介绍 Pentium 4 的微系统结构及其在 SPEC CPU 基准测试程序中的性能表现。Pentium 4 同时也支持多线程,我们将在下一章中讨论这个主题。

Pentium 4 采用了深度乱序推测的微系统结构,我们称这种微系统结构为 Netburst。通过将多发射和高时钟频率结合在一起,Netburst 实现了以获得高指令吞吐量为目标的深度流水。与 Pentium 3 采用的微系统结构类似,前端译码器将每条 IA-32 指令转换为类似典型 RISC 指令的一系列微操作 (uops),并把这些微操作送往动态调度的推测流水线上执行。

同传统的指令 Cache 保存 IA-32 不同, Pentium 4 使用了一种新颖的执行踪迹 Cache 来生成微操作指令流。踪迹 Cache 是一种保存指令序列的指令 Cache,这些序列将被执行,序列中包括由转移分开的非相邻指令;与一般 Cache 不同,踪迹 Cache 试图开发执行指令的时间序列,而不是空间位置;我们将在附录 C 中详细讨论这个问题。

Pentium 4 中的执行踪迹 Cache 是缓存微操作的踪迹 Cache,对应于译码后的 IA-32 指令流。当踪迹 Cache 命中时, Pentium 4 使用执行跟踪缓存使流水线不停顿,从而避免了对 IA-32 指令的重新译码。只有当踪迹缓存缺失时才从二级 Cache 中取指令,并将其译码以重新填补执行踪迹缓存。每时钟周期最多可译码并转换 3 条 IA-32 指令,生成至多 6 条微操作;当一条 IA-32 需要 3 条以上的微操作时,微操作序列将从微代码 ROM 中生成。

执行踪迹 Cache 使用自己的转移目标缓存预测微操作转移的结果。如果执行踪迹 Cache 的命中率足够高(例如,对于 SPEC CPUINT2000 来说,踪迹 Cache 的缺失率低于 0.15%),则意味着基本不再需要对 IA-32 进行译码和取指令操作。

在从执行踪迹 Cache 中取到微操作后,这些微操作将被送往乱序推测流水线执行,这同 2.6 节中的情况类似,只不过使用的是寄存器重命名而不是重排序缓存。每时钟周期内至多可以对 3 条微操作重命名,并将其分发到功能单元队列中。每时钟周期可以提交 3 条微操作。共有 4 个分发端口,通过这 4 个端口,每时钟周期总共可以有 6 条微操作被分发往功能单元。Load 和 store 单元分别拥有各自的端口,另一个端口负责 ALU 运算,而第四个端口负责处理浮点和定点运算。图 2.26 为微系统结构的示意图。

由于 Pentium 4 微系统结构采用动态调度,因此,微操作在执行过程中不会简单地遵循一组静态的流水段。不同的执行流水段(取指令、译码、微操作发射、重命名、调度、执行以及退出)会花费不同的时钟周期数。在 Pentium III 中,执行流水段较以往更长,指令从取指令阶段到执行完毕的这段时间至少为 11 个时钟周期。和其他动态调度流水线一样,如果指令被迫等待操作数,则花费的时间可能更长。我们前面曾经提到, Pentium 4 引入了深度流水线,加上 Pentium III 中的流水段,使得 Pentium 4 获得了更高的时钟频率。当 Pentium 4 于 2000 年首次提出时,贯穿流水线所需的最小时钟周期数增加到了 21 个,处理器的时钟频率为 1.5 GHz。到 2004 年, Intel 推出的新款 Pentium 4 处理器的时钟频率已经达到了 3.2 GHz。为了获得如此高的时钟频率,处理器中又加进了更长的流水线,使得一条简单指令从取指令阶段到执行完毕要花费 31 个时钟周期。这种深度流水线同更快的晶体管速度的结合,使得其时钟频率超过最早的 Pentium 4 处理器多达两倍。

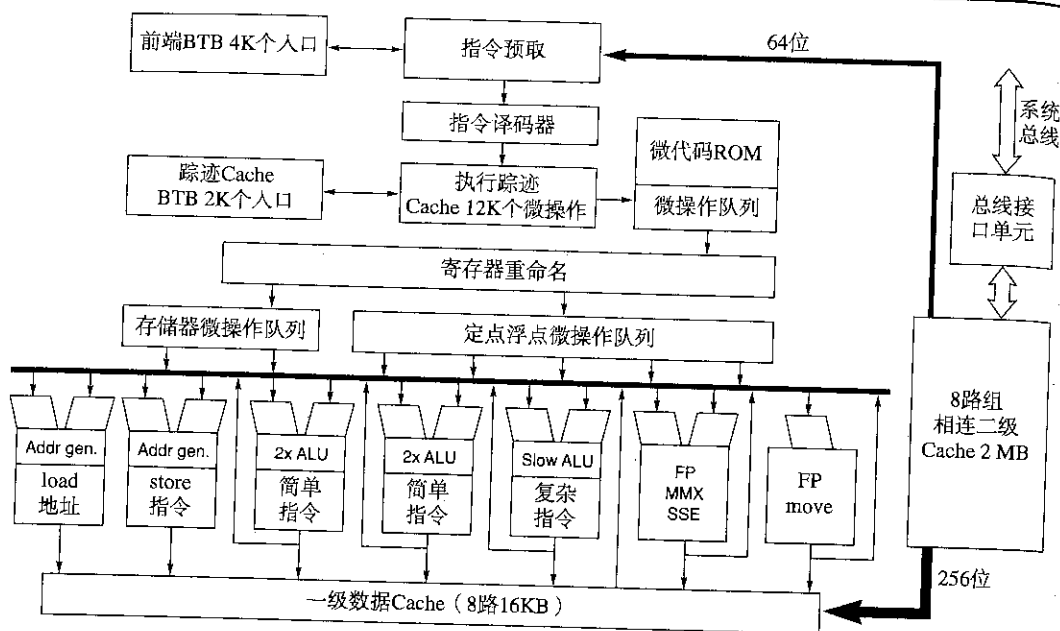


图 2.26 Pentium 4 微系统结构。Cache 容量以 Pentium 4640 为代表。指令一般来源于踪迹 Cache，只有在踪迹 Cache 缺失的情况下才使用前端指令预取单元。此图引自 Boggs 等[2004]

很明显，在深度流水线与如此高的时钟频率组成的环境中，转移预测错误率以及 Cache 缺失率都将相当高。为了把对 DRAM 的访问减少到最小，在微系统结构中引入了一个二级 Cache。转移预测是通过转移目标缓存进行的，该转移目标缓存使用一个包含局部和全局转移历史的二级预测器，大多数最近生产的 Pentium 4 处理器都增加了转移预测缓存的容量，同时改善了静态预测器的性能，这些静态预测器将用于转移目标缓存缺失的情况。图 2.27 中概括了 Pentium 4 微系统结构的几个关键的特性，图下面的文字说明了自 2000 年第一版以来 Pentium 4 所做的改变。

Pentium 4 性能分析

Pentium 4 深度流水线对推测的使用，以及对转移预测的依赖，都是影响性能的关键因素。同时，性能也与存储系统密切相关。尽管可以通过动态调度和大量的外部 load 和 store 指令隐藏 Cache 缺失引起的时延，但是高达 3.2 GHz 的时钟频率意味着，由于要在等待处理缺失的同时填充队列，因此二级 Cache 的缺失仍然有可能引起流水线停顿。

鉴于转移预测和 Cache 缺失的重要性，我们将把注意力集中到这两个领域中。本节中的图表使用 5 个定点 SPEC CPU2000 基准测试程序和 5 个浮点基准测试程序，数据由 Pentium 4 为监测性能而设计的计数器采集。使用的处理器为 Pentium 4640，3.2 GHz 时钟频率，800 MHz 系统总线，667 MHz DDR2 DRAM 主存。

图 2.28 所示为预测错误率，以预测错误的次数/每 1000 条指令表示。需要注意的是，当我们提到流水线性性能时，所指的是预测错误次数/每条指令；同定点基准测试程序相比，在浮点基准测试程序中，转移所占的比例通常较少（48 条转移/每 1000 条指令对 186 条转移/每 1000 条指令），预测准确率也较高（98% 对 94%），其结果如图 2.28 所示，定点基准程序每条指令的预测错误率是浮点基准测试程序的 8 倍。

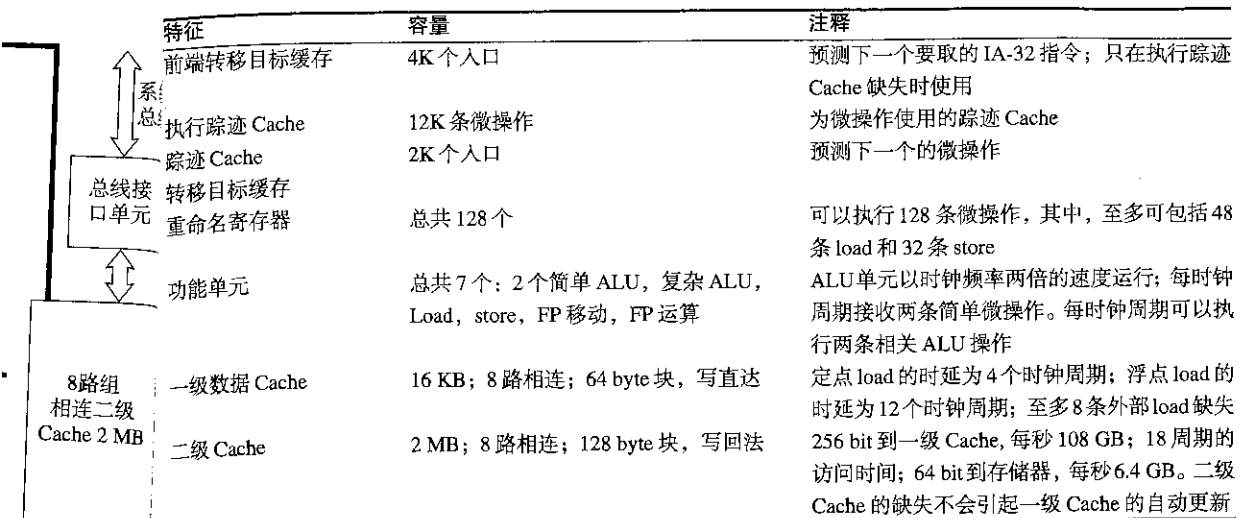


图 2.27 90 nm 工艺实现的 Pentium 4640 (Prescott) 的主要特征。更新的 Pentium 4 使用了容量更大的 Cache 和转移预测缓存, 允许更多的外部 load 和 store, 在存储系统的各层之间有更高的带宽。新型双倍速 ALU 在一个时钟周期内支持背靠背的相关 ALU 运算; 而在其他的设计中, 即使拥有两倍数量的 ALU 也无法拥有这个能力。最初的 Pentium 4 使用 512 个入口的踪迹 Cache BTB, 8 KB 的一级 Cache 和 256 KB 的二级 Cache

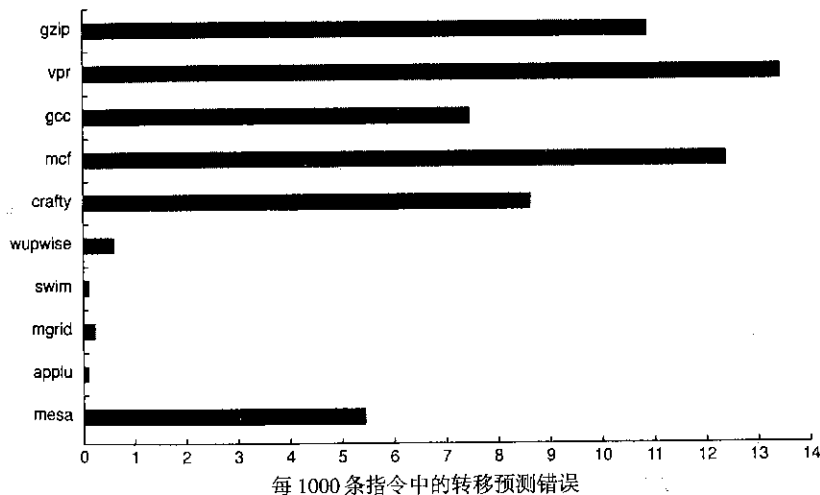


图 2.28 执行 SPEC CPU2000 基准测试程序集中的 5 种定点和 5 种浮点基准测试程序时, 每 1000 条指令中的转移预测错误。此图以及本节中其他图表的数据来自于 Intel 公司的 John Holm 和 Dileep Bhandarkar

由于推测错误的恢复需要时间, 并且会在错误的路径上耗能, 因此转移预测的准确率对推测处理器来说十分关键。图 2.29 所示为根据错误的推测执行的微操作所占的比例。正如我们所预计的, 推测错误率的情况与预测错误率的情况几乎完全相同。

那么 Cache 缺失率在可能的性能损失中又扮演什么角色呢? 对于这组 SPEC 基准测试程序来说, 踪迹 Cache 缺失率几乎可以忽略不计。踪迹 Cache 的缺失率只在一种基准测试程序中 (186.craft) 比

较明显(0.6%)。相比之下,一级和二级Cache的缺失率更加突出。图2.30所示为运行这10种基准测试程序时一级和二级Cache的缺失率。虽然一级Cache的缺失率几乎是二级Cache缺失率的14倍,但是二级Cache的缺失代价却更加严重,同时这种缺失是微系统结构无法回避的,这意味着二级Cache缺失引起的性能损失甚至要超过一级Cache缺失,这种现象在mcf和swim这类基准测试程序中表现得更加突出。

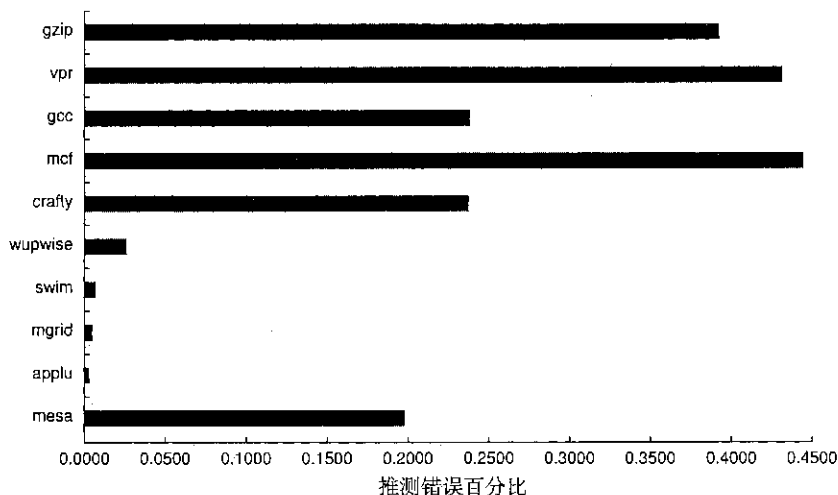


图 2.29 根据错误的推测发射的微操作指令所占的百分比

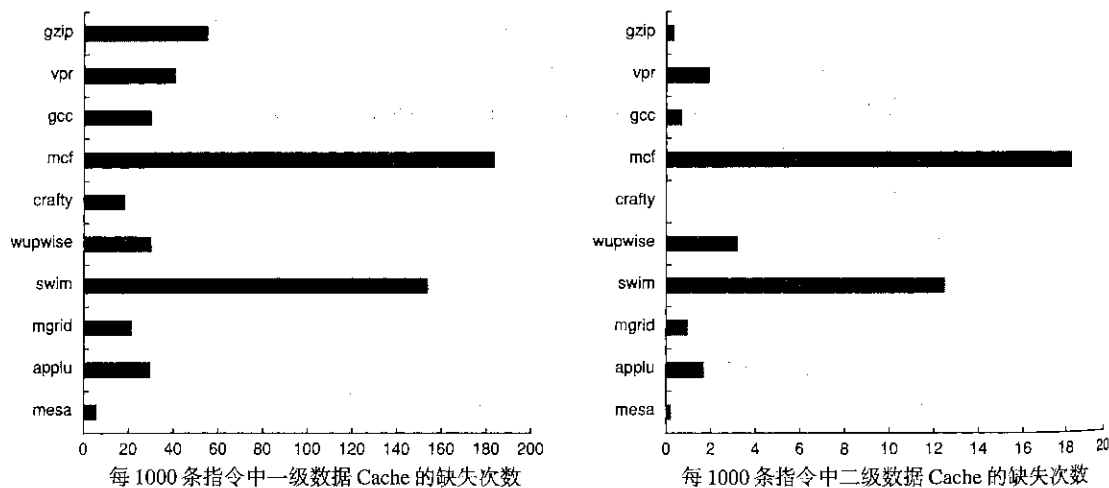


图 2.30 执行10种SPEC CPU2000基准测试程序时,一级数据Cache和二级Cache的缺失次数。一级Cache缺失的标尺是二级Cache缺失的10倍。但是二级Cache的缺失代价至少是一级Cache的10倍,柱体的相对大小表示Cache缺失代价的相对大小。由于无法通过重叠执行隐藏二级Cache缺失引起的延迟,因此二级Cache缺失引起的停顿要多于一级Cache缺失

那么推测错误和Cache缺失的影响又是怎样转化为实际性能损失的呢?图2.31所示为10个SPEC CPU2000基准测试程序的有效CPI。其中有三种较为突出的基准测试程序值得研究:

1. mcf 的 CPI 大约是其 4 种定点基准测试程序的 4 倍。mcf 拥有最差的推测错误率。同时, mcf 一级和二级 Cache 的缺失率在 SPEC 集的所有定点或浮点基准测试程序中也是最差的。mcf 中 Cache 的高缺失率使得处理器不可能隐藏大量的缺失时延。
2. 在 5 种定点基准测试程序中, vpr 的 CPI 是除 mcf 外其他三种的 1.6 倍。这是由于 vpr 的转移预测错误率是定点基准测试程序中最差的(尽管不比平均值低很多), 同时 vpr 的二级 Cache 缺失率也很高, 在定点基准测试程序中仅好于 mcf。
3. swim 的性能在浮点基准测试程序中最差的, 其 CPI 大约是其 4 种浮点基准测试程序平均值的 2 倍多。swim 的问题在于高的一级和二级 Cache 缺失率, 仅好于 mcf。值得注意的是, 尽管 swim 的推测准确率相当出色, 但在推测方面的成功无法隐藏 Cache 特别是二级 Cache 的高缺失率引起的延迟。与之形成鲜明对照的是, 一些基准测试程序的一级 Cache 缺失率较为合理且二级 Cache 的缺失率较低(比如 mgrid 和 gzip), 这使得它们的性能更为出色。

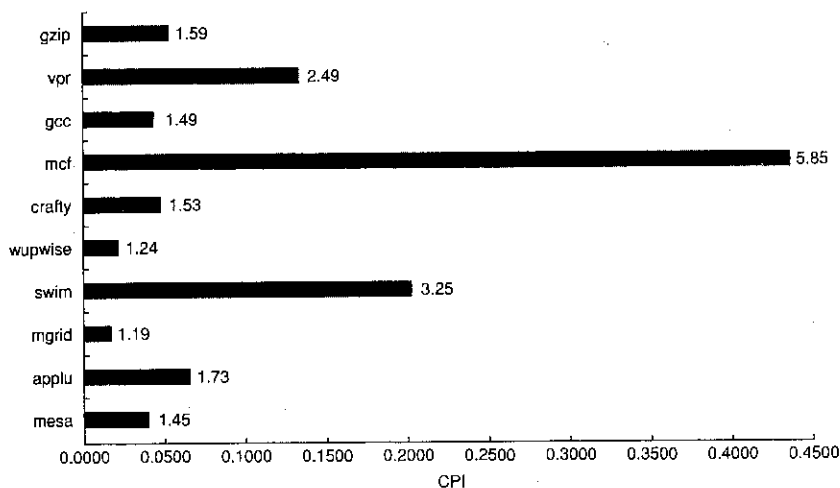


图 2.31 10 种基准测试程序的 CPI。CPI 1.29 倍的增长来源于将 IA-32 指令转换成微操作的过程, 这导致在这 10 种基准测试程序下, 每条 IA-32 指令产生 1.29 个微操作

在本节结束前, 让我们来看看在运行 SPEC 基准测试程序的这组子集时, Pentium 4 和 AMD Opteron 的性能比较。AMD Opteron 和 Intel Pentium 4 有很多相似点:

- 都使用动态调度、推测流水线, 每时钟周期能够发射和提交三条 IA-32 指令。
- 都使用两级片上 Cache 结构, 但是 Pentium 4 使用踪迹 Cache 作为一级指令 Cache, 最新的 Pentium 4 则使用更大容量的二级 Cache。
- 有相似的晶体管数目、核心面积和功耗, 到 2005 年为止, 对时钟频率最快的 Pentium 4 和 AMD Opteron 处理器所做的比较显示, Pentium 4 在这三个方面要高出 AMD Opteron 大约 7%~10%。

Pentium 4 和 AMD Opteron 最明显的不同, 是 Intel Netburst 微系统结构为获得更高的时钟频率而设计的超长流水线。虽然为这两种结构优化的编译器产生的代码序列不尽相同, 但是仍然可以通过 CPI 对这两种处理器进行比较。需要注意的是, 存储结构和流水线结构的不同会对测量方法产生影响, 我们会在第 5 章专门分析存储系统的性能。图 2.32 所示为在 3.2 GHz 的 Pentium 4 上和 2.6 GHz 的 AMD Opteron 上运行 SPEC CPU2000 基准测试程序时, 测得的 CPI。在这个时钟频率下, AMD Opteron 的平均 CPI 要比 Pentium 4 低 1.27。

当然,考虑到Pentium 4的超长流水线,它的平均CPI比AMD Opteron高也在我们的意料之中。问题的关键在于,Netburst用超长流水线结构换来的高时钟频率能否克服高CPI引起的缺陷。到2005年为止,时钟频率最快的Pentium 4和AMD Opteron处理器分别为3.8 GHz和2.8 GHz,通过分析SPEC CPU2000基准测试程序在这两款处理器上的性能表现,我们可以得到上述问题的答案。同图2.32中的情况相比,在图2.33中,3.8 GHz和2.8 GHz的时钟频率会使有效CPI也相应增加,这是Cache缺失率的升高引起的。图2.33使用的基准测试程序与图2.32使用的相同。从图中我们可以看出,Opteron处理器要略微快一些,这意味着Pentium 4的高时钟频率不足以弥补流水线延迟造成的高CPI所带来的缺陷。

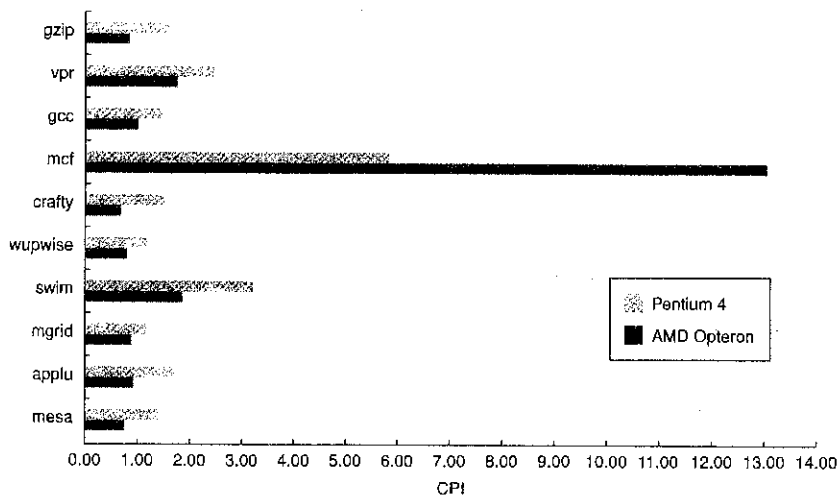


图 2.32 3.2 GHz Pentium 4 的 CPI 是 2.6 GHz AMD Opteron 的 1.27 倍

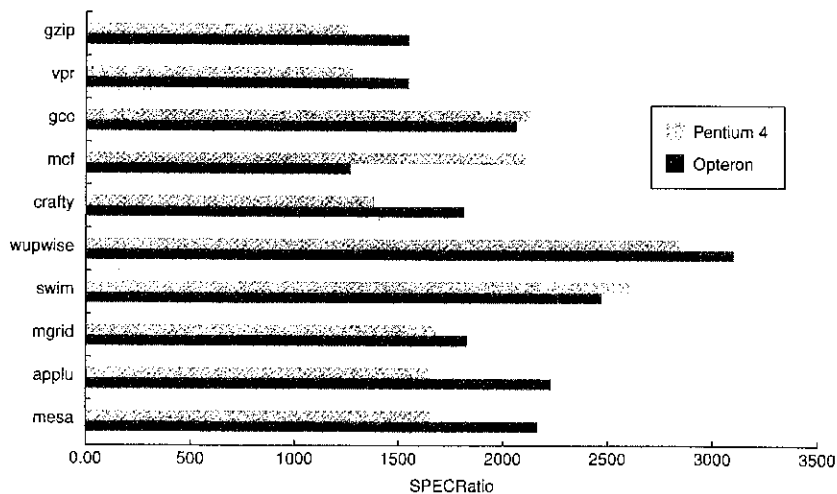


图 2.33 2.8 GHz AMD Opteron 与 3.8 GHz Intel Pentium 4 的性能比较显示 Opteron 在性能上的优势大约为 1.8 倍

因此,尽管Pentium 4的表现也比较出色,但是很明显,通过深度流水得到的高时钟频率和通过多发射获得的高指令吞吐量,并没有像设计师最初期望的那样获得成功。我们将在下一章中深入探讨这个问题。

2.11 谬误和易犯的错误

此处的谬误说明两个问题：第一，片面、简单的断言是不可靠的；第二，基准测试程序的选择会对性能产生影响。

谬误：CPI 越低处理器运行速度越快。

谬误：时钟频率越快处理器运行速度越快。

虽然 CPI 是越低越好，但是复杂的多发射流水线的时钟频率通常要低于简单流水线的时钟频率。在指令级并行度有限或是指令级并行性无法被硬件有效利用的情况下，更快的时钟频率往往会更成功。但是在有大量指令级并行性存在的情况下，能够充分利用指令级并行性的处理器往往表现更加出色。

IBM 的 Power 5 是为高性能定点和浮点运算设计的；它包含两个处理器核心，每个处理器上可以实现每时钟周期处理 4 条指令的速率，包括两条浮点指令和两条 load-store 指令。到 2005 年为止，Power 5 处理器的最快时钟频率为 1.9 GHz。而作为对比，Pentium 4 包含单独的一个支持多线程（见下一章）的处理器核心，处理器可以在超长流水线上实现每时钟周期 3 条指令的速率，到 2005 年为止，Pentium 4 的最快时钟频率为 3.8 GHz。

因此，如果 Power 5 的指令数和 CPI 的乘积低于 Pentium 4 的一半，则 Power 5 要比 Pentium 4 更快些。如图 2.34 所示，在浮点程序中，CPI × 指令数为 Power 5 带来了明显的性能优势，有时这种优势甚至能达到两倍多。而在定点程序中，Power 5 在 CPI × 指令数方面具有的优势往往不足以弥补它在时钟频率上与 Pentium 4 的差距。通过对比我们发现，在浮点程序中，Power 5 在 CPI × 指令数上的优势为 3.1 倍，但是在定点程序中这一优势仅为 1.5 倍。2005 年，Pentium 4 的最快时钟频率正好是 Power 5 的两倍，运行 SPECfp2000 时，Power 5 要比 Pentium 4 快 1.5 倍，而在运行 SPECint2000 时，Pentium 4 比 Power 5 快 1.3 倍。

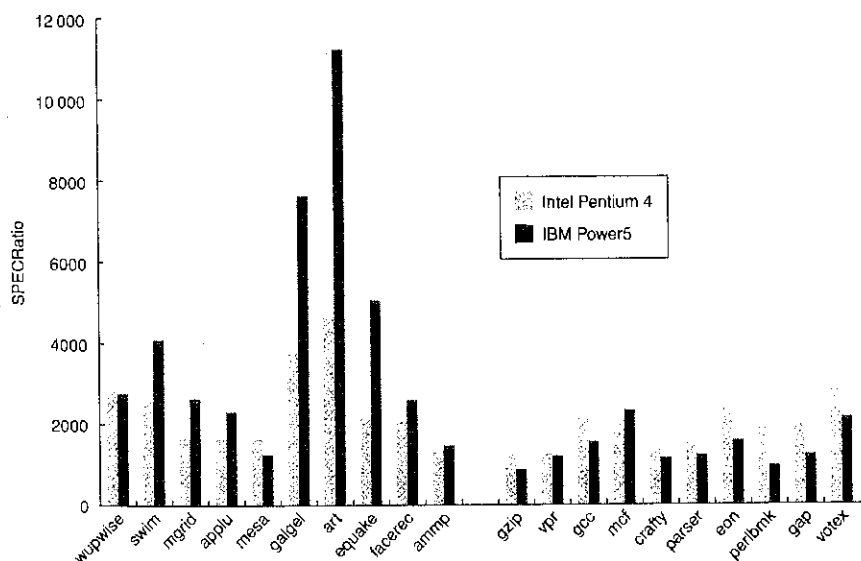


图 2.34 1.9 GHz IBM Power 5 与 3.8 GHz Intel Pentium 4 在 20 个 SPEC 基准测试程序上的比较（左侧为 10 个定点基准测试程序，右侧为 10 个浮点基准测试程序），这个比较表明在定点工作负载中，时钟频率更高的 Pentium 4 通常更快，而在浮点工作负载中，CPI 较低的 Power 5 通常更快

易犯的错误:有时候大一点、笨一点反而更好。

为了改善 CPI, 高级流水线已经将注意力集中到那些新颖的、日益复杂的技术上。21264 使用的复杂 Tournament 预测器总共有 29K bit, 而早期的 21164 使用的 2 bit 预测器只有 2K 个入口(总共 4K bit)。在 SPEC95 基准测试程序上, 更复杂的预测器在除一项之外的所有项上的表现都超过了简单 2 bit 预测器。在 SPECint95 中, 平均每 1000 条提交指令中, 21264 的预测错误为 11.5 个, 而 21164 的预测错误为 16.5 个。

但令人惊讶的是, 简单 2 bit 方法在事务处理工作负载上的表现居然要好于复杂的 21264 方法(17 个错误预测/每 1000 条完成指令对 19 个错误预测/每 1000 条完成指令)。一个 bit 数仅为复杂方法 1/7 的简单方法怎么会在实际工作中表现得更加出色呢? 答案在于工作负载的结构。事务处理工作复杂的代码量非常庞大(远远超过所有 SPEC 95 基准测试程序), 同时转移频率也相当高。而 21164 基于局部行为的转移预测器是 21264 的两倍(21164 中的 2K 对 21264 中的 1K), 看起来是这方面的优势在起作用。

这个易犯的错误提醒我们, 不同的应用程序可能会表现出不同的性能。随着处理器越来越复杂, 一些微系统结构的特征都是针对某种特定类型的程序设计的, 在这种情况下, 不同的应用将更多地表现出不同的性能。

2.12 结论

由于希望在不改变标准的单处理器编程模型的前提下提高性能, 因此多发射组织结构引起了人们的巨大兴趣。虽然利用指令级并行度从概念上看比较简单, 但是在实际设计过程中所遇到的问题却异常复杂。如果只凭简单的分析, 我们也许会对性能有很高的期望, 但实际上要想达到这种期望是非常困难的。

过去十年的研究大多集中在提高多发射处理器的时钟频率以及努力缩短持续性能与性能峰值之间的距离上, 而不是在微系统结构上开拓全新的方法。同 1995 首次出现的动态调度、多发射处理器相比, 过去五年中出现的动态调度、多发射处理器(Pentium 4, IBM Power 5, AMD Athlon 和 Opteron)在基本结构与持续指令发射率(每时钟周期 3~4 条指令)上并没有本质的改进! 只不过是时钟频率提高了 10~20 倍, Cache 容量提高了 4~8 倍, 重命名寄存器的数量增长了 2~4 倍, load-store 单元的数量增长了 2 倍。因此性能才得以提高了 8~16 倍。

多发射能够在快速提升的时钟频率和恶化的 CPI 之间起到折中作用, 但是这种折中是很难量化的。在本书的 1995 版中, 我们曾经提到:

虽然可以设计一个有很高时钟频率的高级多发射处理器, 但时钟频率的 1.5 和 2 的因子使得最高时钟频率的处理器和最高级的多发射处理器区别开来。造成这一结果的原因到底是由于两个方案在基本实现中的差别, 多发射处理器的复杂度, 还是缺乏实现上述处理器的经验? 想把这些缘由搞清楚还为时过早。

通过对 3.8 GHz Pentium 4 的分析, 我们开始逐渐清晰地认识到, 对如何设计这类处理器的理解才是产生这些限制的主要原因。但是我们还不清楚, 在可用指令级并行度、指令级并行度的开发效率以及功耗等因素的限制下, 这种通过每时钟周期发射 3~4 条指令以使处理器获得更高时钟频率的方法是否能获得更大的成功。我们会在下一章中研究这个问题。此外, 通过对 Opteron 和 Pentium 4 的比较我们发现, 虽然通过超长流水(20~30 个流水段)可以获得更快的时钟频率, 但是由此引起的额外流水线停顿却严重损失了高时钟频率的性能优势。我们将在下一章分析这个问题。

从1995年到2005年,有一点变得越来越清晰,那就是在多发射处理器中,持续性能与性能峰值的比通常都相当高,而且这个值一般情况下会随发射率的上升而增长。对Power 5和Pentium 4以及Pentium 4和Pentium III的比较提醒我们,想要评价时钟周期和CPI的关系,或是在流水线深度、发射率和其他特性之间做出折中,是非常困难的。

很明显,我们需要探索其他的方法。Pentium 4的高时钟频率版本已经被弃用。在Power4和Power 5中,IBM开始在一个芯片上尝试使用双核技术,而Intel和AMD也都推出了双核的早期版本。我们将下一章继续讨论这个主题,并解释为什么对指令级并行度长达20年的追求即将结束。

2.13 历史回顾和参考文献

随书光盘上的K.4节介绍了流水线和指令级并行的发展历程。我们为深入阅读和探讨这些主题提供了参考文献。

2.14 范例分析及习题^①

范例分析 1: 探讨微系统结构技术的影响

通过这个范例阐明以下概念:

- 基本指令调度、重排序和分发
- 多发射和冒险
- 寄存器重命名
- 乱序和推测执行
- 在哪里花费乱序资源

你被要求设计一个新的处理器微系统结构,你需要将你的硬件资源最优化分配。你应当应用哪些在第2章中学习到的技术?已知各种功能单元和存储器的时延,同时你还得到了一些具有代表性的代码。你的老板并没有对你的设计提出明确的性能要求,但是凭经验你知道,在所有情况下都一样,当然是越快越好。让我们从基础开始,图2.35提供了一段指令以及时延列表。

- 2.1 [10]<1.8, 2.1, 2.2>在图2.35所示的代码序列中,如果在前序指令完成执行之前,所有的新指令都不能开始执行,那么这段代码的性能(每次循环迭代所花的时钟周期)底线是多少?忽略前端取指令和译码。假设执行不会因为下一条指令的缺失而引起停顿,但是每时钟周期只能发射一条指令。假设转移被选中,并且转移时延为1个时钟周期。
- 2.2 [10]<1.8, 2.1, 2.2>思考延迟数的真正含义——时延数表明一个给定的功能在生产输出的过程中所花费的时钟周期数,仅此而已。如果整体流水线为每个功能单元的时延周期停顿,那么这样至少可以保证所有的背靠背指令对(一个生产者紧跟一个消费者)正确执行。但是并不是所有的指令对都有生产者/消费者的关系。有时两条相邻的指令之间没有任何关系。如果流水线只在检测到真数据相关的情况下停顿,而不是只因功能单元忙就盲目地停顿,那么在这种情况下,图2.35所示代码中的循环体需要多少个时钟周期?在代码中在需要停顿的地方插入<stall>,并给出时延(提示:时延为“+2”的指令需要在代码序列中插入2个<stall>时钟周期。试想:一个1周期的指令时延为1+0,即没有额外的等待状态。所以时延为1+1意味着1个停顿周期;1+N有N个停顿周期)。

^① 本范例分析由 Robert P. Colwell 提供。

			超出一个时钟周期的时延	
Loop: LD	F2,0(Rx)		Memory LD	+3
I0: MULTD	F2,F0,F2		Memory SD	+1
I1: DIVD	F8,F2,F0		Integer ADD, SUB	+0
I2: LD	F4,0(Ry)		Branches	+1
I3: ADDD	F4,F0,F4		ADDD	+2
I4: ADDD	F10,F8,F2		MULTD	+4
I5: SD	F4,0(Ry)		DIVD	+10
I6: AODI	Rx,Rx,#8			
I7: ADDI	Ry,Ry,#8			
I8: SUB	R20,R4,Rx			
I9: BNZ	R20,Loop			

图 2.35 习题 2.1 到习题 2.6 使用的代码和时延

- 2.3 [15]<2.6, 2.7>思考一个多发射设计。假设你有两条执行流水线，每条流水线每时钟周期可以开始执行一条指令，并且前端有足够的取指令/译码带宽，因此不会使执行产生停顿。假设结果可以从一个执行单元立即提交给另一个执行单元或这个执行单元自身。并进一步假设真数据相关是使流水线停顿的唯一原因。那么在这种情况下，该循环需要多少个时钟周期？
- 2.4 [10]<2.6, 2.7>在习题 2.3 所述的多发射设计中，你会发现一些微小的问题。即使这两条流水线有完全相同的指令系统，它们也不是完全相同、可交换的，这是由于它们之间的顺序必须反映原始程序中的指令顺序。如果在指令 $N+1$ 在流水线 1 中开始执行的同时，指令 N 在流水线 0 中开始执行，并且 $N+1$ 的执行时延比 N 短，那么 $N+1$ 就有可能在 N 之前执行完成（即使程序顺序中的情况不是这样）。请给出至少两个理由，解释这种情况为什么是冒险的，以及为什么需要在微系统结构中专门考虑这个问题。在图 2.35 所示的例子中找出两条代码，证明这种冒险。
- 2.5 [20]<2.7>对图 2.35 的代码进行重排序以提高其性能。假设使用习题 2.3 所述的双流水线机器，并假设习题 2.4 所述的乱序问题能够得到妥善解决。现在只需考虑真数据相关和功能单元时延。重排序后的代码需要花费多少个时钟周期？
- 2.6 [10/10]<2.1, 2.2>在流水线中，如果在一个时钟周期内没有初始化一项操作，那么就等于浪费了一次机会，或者说你的硬件没有充分发挥它的潜能。
- [10]<2.1, 2.2>代码在按照习题 2.5 的要求重排序后，将两条流水线都考虑在内，所有时钟周期中被浪费（没有新的操作被初始化）的周期所占的比例为多少？
 - [10]<2.1, 2.2>为了提高性能并将错失并行机会的次数控制到最少，循环展开是一种在代码中寻找更多指令级并行度的标准编译技术。
 - 将习题 2.5 中重排序的代码展开为两个迭代，速度提升了多少？（在本题中，将第 $N+1$ 个迭代的指令表示为绿色以区别第 N 个迭代的代码；在实际过程中展开循环需要重新为寄存器赋值以避免迭代之间的冲突）。
- 2.7 [15]<2.1>计算机将大部分时间都花费在执行循环上，多循环迭代是一种使 CPU 资源始终保持工作状态的好方法。但是这并不像说起来那么容易；编译器只发射一个循环代码的副本，因此即使多个循环体处理的是不同的数据，它们也有可能使用相同的寄存器。为了使多迭代体不会发生寄存器冲突，需要重命名寄存器。图 2.36 所示为需要重命名的示例代码。

编译器可以简单地将循环展开,并通过使用不同的寄存器避免冲突,但是如果我们期望硬件展开循环,那么必须使用寄存器重命名。那么怎样重命名呢?假设你的硬件有一个临时寄存器组(称为T寄存器,假设有64个,从T0到T63),可以使用它们替代编译器指定的寄存器。这些重命名硬件由源寄存器目标索引,表中的值为上一次指向该寄存器的目标的T寄存器(将这些表值想象为生产者,源寄存器为消费者;只要能让消费者找到结果,生产者将结果放在哪里并不重要)。考虑图2.36所示的代码。每当在代码中看到目标寄存器时,从T9开始,替换下一个可用的T。之后更新所有相应的源寄存器,从而维护真数据相关。写出结果代码(提示:见图2.37)。

```

Loop: LD    F2,0(Rx)
I0:  MULTD  F5,F0,F2
I1:  DIVD   F8,F0,F2
I2:  LD     F4,0(Ry)
I3:  ADDD   F6,F0,F4
I4:  ADDD   F10,F8,F2
I5:  SD     F4,0(Ry)

```

图 2.36 寄存器重命名习题的示例代码

```

I0:  LD     T9,0(Rx)
I1:  MULTD  T10,F0,T9
...

```

图 2.37 寄存器重命名期望的输出

- 2.8 [20]<2.4>习题2.7探讨了简单寄存器重命名的问题:当硬件重命名被视为源寄存器时,它替代了上一条指令指向的源寄存器的目标T寄存器。当重命名表发现源寄存器时,它为目标寄存器替换下一个可用T。超标量设计每时钟周期要在机器的各段处理多条指令,包括寄存器重命名。一个简单的标量处理器为每条指令查找源寄存器映射,并且每时钟周期分配一个新的目标映射。超标量处理器同样必须完成这些工作,同时还要保证正确处理当前两条指令之间的目标到源的关系。考虑图3.28中的代码序列。假设我们要同时对前两条指令进行重命名。进一步假设在这两条指令将要被重命名的时钟周期开始之前,能够确定将要使用的后面两条可用T寄存器。从概念上讲,我们需要做的是为第一条指令查找重命名表,之后为指令目标的每个T寄存器更新表。对第二条指令也是一样,这样内部的指令相关就可以被正确处理。但是如果想在同一个时钟周期内既将T寄存器目标写进重命名表,又为第二条指令查找重命名表,时间肯定是不够用的。因此寄存器替换必须要动态完成(与更新寄存器重命名表并行)。图2.39所示为一个循环,可以使用多路器和比较器动态完成寄存器重命名。你的任务是为代码中的每一条指令写出一个循环接一个循环的重命名表的状态。假设表从每个等于其索引的入口开始(T0=0; T1=1, ...)。

```

I0:  MULTD  F5,F0,F2
I1:  ADDD   F9,F5,F4
I2:  ADDD   F5,F5,F2
I3:  DIVD   F2,F9,F0

```

图 2.38 超标量寄存器重命名的示例代码

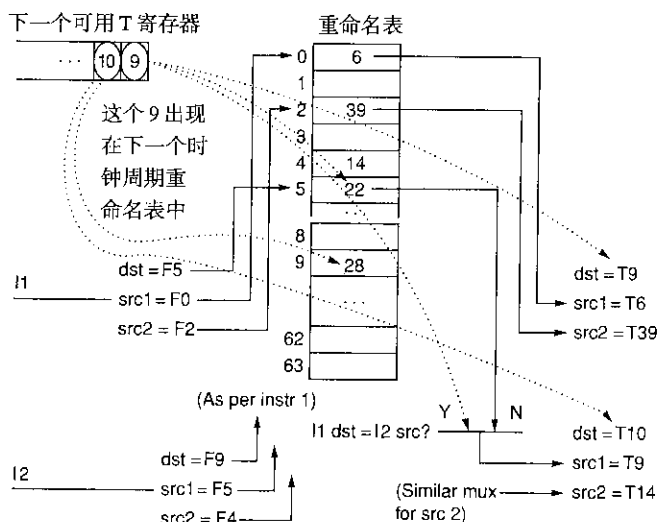


图 2.39 超标量机器的重命名表和动态寄存器替换逻辑（其中“src”为源，“dst”为目标）

- 2.9 [5]<2.4>寄存器重命名到底需要做什么？如果这个问题一直使你感到困惑的话，不妨回头看看你正在执行的汇编代码，想想为了获得一个正确的结果我们需要做些什么。例如，考虑三路超标量机器同时对下面三条指令重命名：

```
ADDI    R1, R1, R1
ADDI    R1, R1, R1
ADDI    R1, R1, R1
```

如果 R1 的值从 5 开始，那么当这个序列执行完成时 R1 的值又是多少？

- 2.10 [20]<2.4, 2.9>为使用寄存器制定相关的系统结构规则，VLIW的设计者可以有几种选择。假设一个有自动执行流水线的 VLIW 设计：一旦操作被初始化，它的结果最晚在 L 个时钟周期后将出现在目标寄存器中（ L 是该操作的时延）。由于永远不会有充足的寄存器，因此必须尽可能地利用现有的寄存器。考虑图 2.40。如果 load 指令的时延为 $1 + 2$ 个时钟周期，将这个循环展开一次，在没有任何流水线中断或延迟的情况下，写出每时钟周期能够处理两条 load 和两条加法运算的 VLIW 怎样将对寄存器的使用减少到最小。在自动流水线中，给出一个干扰流水线和导致错误结果的例子。

```

Loop: LW    R1,0(R2) ; LW    R3,8(R2)
      <stall>
      <stall>
      ADDI  R10,R1,#1; ADDI  R11,R3,#1
      SW    R1,0(R2) ; SW    R3,8(R2)
      ADDI  R2,R2,#8
      SUB   R4,R3,R2
      BNZ   R4,Loop
  
```

图 2.40 有两条 add、两条 load 和两条 stall 的 VLIW 示例代码

- 2.11 [10/10/10]<2.3>假设一个五段单流水线微系统结构（取指令、译码、执行、存储器、写回），使用图 2.41 中的代码。除 LW 和 SW 外的所有的操作均为 1 个时钟周期，延迟为 $1 + 2$ 个

时钟周期, 转移延迟为 $1 + 1$ 个时钟周期。不考虑提交。以一个时钟周期为一个阶段, 写出一个循环迭代中的各条指令所经过的各个阶段。

```

Loop: LW    R1, 0(R2)
      ADDI   R1, R1, #1
      SW     R1, 0(R2)
      ADDI   R2, R2, #4
      SUB    R4, R3, R2
      BNZ    R4, Loop
  
```

图 2.41 习题 2.11 的循环代码

- a. [10]<2.3>每个循环迭代中由于转移开销损失的时钟周期为多少?
- b. [10]<2.3>假设一个静态转移预测器能够在译码阶段识别后面的转移。那么在这种情况下由于转移开销而损失的时钟周期又是多少?
- c. [10]<2.3>假设有一个动态转移预测器。在正确预测的情况下损失的时钟周期数是多少?
- 2.12 [20/20/20/10/20]<2.4, 2.7, 2.10>让我们来考虑动态调度的情况。假设微系统结构如图 2.42 所示。假设 ALU 可以执行所有数学运算 (MULTD, DIVD, ADDD, ADDI, SUB) 和转移, 并且每时钟周期内保留站 (RS) 至多可以为每个功能单元分配一条操作 (每个 ALU 一个操作, 加上 LD/ST 单元的一个存储器操作)。

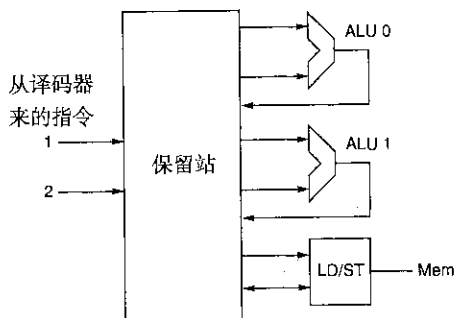


图 2.42 乱序微系统结构

- a. [15]<2.4>假设图 2.35 所示序列中的所有指令都出现在 RS 中, 没有进行寄存器重命名。在代码中标出可通过寄存器重命名改善性能的指令。提示: 注意 RAW 和 WAW 冒险。假设所有的功能单元的时延都如图 2.35 所示。
- b. [20]<2.4>假设经过(a)后, 经过寄存器重命名之后的代码在时钟周期 N 进驻 RS, 时延情况如图 2.35 所示。以 1 个时钟周期为单位, 写出 RS 乱序分配指令的过程, 使代码能够获得最优性能 (假设对 RS 的限制与(a)相同。结果必须在可用前写入 RS, 即不能旁路)。代码序列将花费多少个时钟周期?
- c. [20]<2.4>在(b)中, 我们尝试了对这些指令进行最优调度。但是在实际过程中, 感兴趣的指令队列通常不会全部出现在 RS 中。在实际过程中, 很多情况下 RS 会被清空, 当新的代码序列流来自译码器时, RS 必须选择分发已有的指令。假设 RS 是空的。在时钟周期 0 时, 这段代码中前两条寄存器重命名指令出现在 RS 中。假分配操作需要花费 1 个时钟周期, 并且功能单元的时延情况如习题 2.2 中所示。进一步假设前端 (译码/寄存器重

- 命名)可以保持每时钟周期提供两条新指令的速度。以1个时钟周期为单位,写出RS分配的顺序。现在这段代码序列花费的时钟周期数又是多少?
- d. [10]<2.10>如果你想进一步改进(c)中的结果,有以下方法可供选择:(1)另一个ALU;(2)另一个LD/ST单元;(3)完全旁路ALU的结果,将结果直接交给后继操作;(4)将时延减半。上述方法哪些能够起到改进作用?哪种方法最有效?
- e. [20]<2.7>现在让我们来考虑推测的问题,越过一个或多个条件转移进行取指令、译码和执行的动作。我们之所以要这样做,主要有两个原因:在(c)中使用的分配调度的方法会产生大量的微操作,而我们已经知道计算机的大部分时间都花费在循环上(这意味着使程序回到循环顶部的转移的可预测性是非常高的)。循环告诉我们到哪里去寻找更多的工作;稀疏分配调度使我们可以提早完成一些这样的工作。在(d)中,你可以通过分析循环找到程序的关键路径。试将该路径隐含。在(d)中,为了使两个循环值得工作,需要多少个额外的时钟周期(假设所有的指令都已进驻RS)?假设所有的功能单元都能充分流水。

范例分析 2: 对转移预测器建模

通过这个范例阐明以下概念:

● 对转移预测器建模

为了真正理解计算机系统结构,除学习微系统技术外,还必须对计算机编程。实际动手对各种微系统结构建模是一个不错的方法。写一段C或Java程序对一个2,1转移预测器进行建模。你的程序需要从名为 history.txt 的文件中读取信息(参见随书光盘,见图 2.43)。

0x40074cdb	0x40074cdf	1
0x40074ce2	0x40078d12	0
0x4009a247	0x4009a2bb	0
0x4009a259	0x4009a2c8	0
0x4009a267	0x4009a2ac	1
0x4009a2b4	0x4009a2ac	1
...		

↑
转移指
令地址

↑
转移目
标地址

↑
1:选中
0:不选中

图 2.43 输入文件 history.txt 的格式示例

文件的每一行有三项数据,由制表符分开。每行的第一列是16进制的转移指令地址。第二列是16进制的转移目标地址。第三列为1或0;1表示被选中的转移指令,0表示未被选中的转移指令。你的模型需要考虑的转移总数就是文件的行数。假设有一个直接映射的BTB,不用考虑指令长度和组合(即如果BTB有4个入口,在0x0, 0x1, 0x2和0x3的转移指令将分别进驻这4个入口,但是0x4的转移指令将覆盖BTB[0])。对输入文件的每一行,你的模型将读取一对数据值,每当对转移预测器建模时调整各种表,并收集关键的性能统计。你的程序的最终输出应当如图 2.44 所示。

模型中BTB的入口数目应当由命令行输入。

BTB 命中的次数: 54 390, 转移总数: 55 493, 命中率: 99.8%

预测错误次数: 1562 / 55493, 2.8%

<一个简单的 Unix 命令行脚本中就包含有最常见的转移，在这里写出你是如何得到这个转移的…>

最常见的转移在总共 55 493 个转移中出现了 15 418 次; 占 27.8%

最常见的转移 = 0x8484ef, 预测正确次数 = 19 151 (预测正确总数为 36 342) 或所占比例为 52.7%

转移缺失总数: 121

强制缺失总数: 104

$$\text{总容量缺失} = \text{总缺失} - \text{强制缺失} = 17$$

BTB 命中总数: 54 390, 转移总数: 55 493, 命中率: 99.8%

预测错误次数: 1103 / 54 493, 2.0%

BTB 长度	预测错误率
1	0.0000
2	0.0000
3	0.0000
4	0.0000
5	0.0000
6	0.0000
7	0.0000
8	0.0000
9	0.0000
10	0.0000
11	0.0000
12	0.0000
13	0.0000
14	0.0000
15	0.0000
16	0.0000
17	0.0000
18	0.0000
19	0.0000
20	0.0000
21	0.0000
22	0.0000
23	0.0000
24	0.0000
25	0.0000
26	0.0000
27	0.0000
28	0.0000
29	0.0000
30	0.0000
31	0.0000
32	0.0000
33	0.0000
34	0.0000
35	0.0000
36	0.0000
37	0.0000
38	0.0000
39	0.0000
40	0.0000
41	0.0000
42	0.0000
43	0.0000
44	0.0000
45	0.0000
46	0.0000
47	0.0000
48	0.0000
49	0.0000
50	0.0000
51	0.0000
52	0.0000
53	0.0000
54	0.0000
55	0.0000
56	0.0000
57	0.0000
58	0.0000
59	0.0000
60	0.0000
61	0.0000
62	0.0000
63	0.0000
64	0.0000
65	0.0000
66	0.0000
67	0.0000
68	0.0000
69	0.0000
70	0.0000
71	0.0000
72	0.0000
73	0.0000
74	0.0000
75	0.0000
76	0.0000
77	0.0000
78	0.0000
79	0.0000
80	0.0000
81	0.0000
82	0.0000
83	0.0000
84	0.0000
85	0.0000
86	0.0000
87	0.0000
88	0.0000
89	0.0000
90	0.0000
91	0.0000
92	0.0000
93	0.0000
94	0.0000
95	0.0000
96	0.0000
97	0.0000
98	0.0000
99	0.0000
100	0.0000

1	32.94%
2	6.42%
4	0.28%
8	0.23%
16	0.21%
32	0.20%
64	0.20%

图 2.44 程序输出格式示例

2.13 [20/10/10/10/10/10/10] <2.3> 写一段程序对一个64个入口的简单四状态转移目标缓存建模。

- [20]<2.3>BTB 的总体命中率是多少（转移到 BTB 中被找到的次数所占的比例）？
- [10]<2.3>冷启动的总体转移预测错误率是多少（正确预测转移的次数所占的比例，不管预测是否属于该转移）？
- [10]<2.3>找出最常见的转移。对该转移的正确预测在预测正确的总数中占多少（提示：统计出该转移在 history.txt 文件中出现的次数，跟踪该转移的各个实例在 BTB 模型中的花费）？
- [10]<2.3>你的转移预测器遇到了多少次容量缺失的情况？
- [10]<2.3>比较冷启动和热启动的效果。使用相同的输入数据建立历史表，并收集统计信息。
- [10]<2.3>冷启动 BTB 4 多次，BTB 大小为 16, 32 和 64。画出导致的 5 个预测错误率，并画出 5 个命中率。
- [10]提交经过注释的、完成的代码。

第3章 指令级并行性的限制

目前处理器正朝着开发出更多指令级并行操作的方向发展……

J. Fisher [1981], 摘自引入“指令级并行”的论文

尽管 IA-64 标榜其 EPIC 系统结构比超标量更加简单, 但令人吃惊的是, 它并没有宣称高时钟频率。其原因不得而知, 但可以预见的是, 致力于改进 CPI 的 IA-64 系统结构的整体复杂度仍然较高, 这使它难以提高时钟频率。

M. Hopkins [2000], 对 IA-64 系统结构的介绍。IA-64 是 HP 和 Intel 共同开发的新一代系统结构, 旨在保证系统结构 simplicity, 同时开发出更多的指令级并行性以提高性能。

3.1 介绍

在上一章中曾经提到, 在从 20 世纪 80 年代中期开始的这 20 年里, ILP 一直是处理器设计要考虑的首要因素。在最初的 15 年里, 我们见证了一系列越来越复杂的流水线机制、多发射机制、动态调度机制和推测机制。而从 2000 年开始, 设计者们开始将精力集中在优化设计上, 或尝试在不改变发射速率的情况下获得更快的时钟频率上。正如我们在上一章结束时所提到的, 开发指令级并行的时代即将结束。

在本章的开始, 我们将首先讨论指令级并行的限制, 这些限制来自程序结构、硬件的预算以及推测使用的关键技术(如转移预测等)的准确率。在 3.5 节, 我们将讨论线程级并行, 将其作为指令级并行的替代或补充。最后, 我们将从性能和效率两方面对一组当前的典型处理器进行比较, 以此作为本章的总结。

3.2 指令级并行性限制的研究

开发指令级并行的历史最早可以追溯到 20 世纪 60 年代的首款流水线处理器。在 20 世纪 80 年代和 90 年代, 开发指令级并行是使性能迅速提升的关键技术。程序中到底存在多少指令级并行度? 从长远来看, 这个问题是决定性能提升速度能否跟上集成电路技术发展速度的关键。而在近期内如何开发出更多的指令级并行度是一个关键问题, 这对于计算机设计者和开发编译器的程序员们来说都很重要。本节的数据为我们提供了一种用来评价上一章所介绍技术有效性的手段, 这些技术包括消除存储器二义性、寄存器重命名以及推测等。

在本节中, 我们将简要回顾对这些问题的研究历程。附录 K 的历史回顾一节介绍了一些这方面的研究, 包括本节用到的数据的来源(Wall 在 1993 年的研究)。所有对可用指令级并行度的研究都可以概括为: 首先设定一组假设, 然后研究在这个假设环境中可用的指令级并行度。我们使用的数据来源最接近于实际的情况; 但实际上, 这个硬件最终还是不大可能实现。尽管如此, 所有的这

类研究都假设编译技术能够达到一定水平,其中的一些假设有可能影响程序的结果,功能再强大的硬件也无法避免。

未来编译技术的发展和全新的硬件技术也许能解决当前研究中存在的问题;但是,从目前看来,仅凭现有的硬件技术和编译技术在短期内恐怕难以做到。例如,在上一章中介绍的值预测技术,虽然它能消除数据相关的问题,但是要想对性能产生显著影响,现有的准确率是远远不够的。实际上,鉴于某些原因(将在3.6节中讨论),我们可能已经达到了开发指令级并行度的极限。这一节将作为过渡,有助于我们深入理解这个问题。

硬件模型

为了搞清楚什么是指令级并行性的限制,我们首先需要定义一个理想处理器。理想处理器是指消除了所有指令级并行性约束的处理器。在理想处理器中,对指令级并行的唯一约束来自寄存器或存储器中的实际数据流。

理想的处理器应具备以下假设:

1. **寄存器重命名**: 可用虚寄存器数量没有限制,因此可以避免所有的WAW和WAR冒险,并且可以有无限的指令同时开始执行。
2. **转移预测**: 最佳的转移预测,能够准确预测所有的条件转移。
3. **跳转预测**: 能够准确预测所有的跳转(包括用于返回的跳转寄存器和需要计算的跳转)。最佳跳转预测和最佳转移预测的组合,相当于使处理器拥有最佳的推测能力,并且拥有无限容量的指令缓存。
4. **存储器地址别名分析**: 能够准确确定所有的存储器地址,在不引用同一地址的情况下,load指令可以移动到store指令之前。需要注意的是,这实现了最佳的地址别名分析。
5. **最优Cache**: 所有的存储器访问花费1个时钟周期。在实际情况中,超标量处理器通常在隐藏Cache缺失的过程中会损失大量的指令级并行度;因此最优Cache会大大优化处理器的性能。

上述假设2和假设3消除了所有的控制相关性。同时,假设1和假设4消除了除真数据相关之外的所有相关性。以上这4条假设意味着:经过调度,程序执行过程中的任意一条指令都可以在其前序相关指令执行完成之后的时钟周期立即开始执行。在这组假设下,程序中动态调度的最后一条指令甚至可以被调度为在第一个时钟周期执行!同时,这组假设还意味着对控制和地址的推测都是完美的。

我们首先讨论无限发射且可以任意超前计算的处理器模型。我们将要讨论的所有处理器模型均没有限制哪些类型的指令可以在一个时钟周期内执行。在无限发射的模型中,这意味着可以在一个时钟周期内发射无限条load或store指令。此外,还假设所有功能单元的时延为1个时钟周期,因此,所有的相关指令序列都可以在相继的时钟周期内被连续地发射。而超过1个时钟周期的时延可能会减少每时钟周期发射的指令数量,尽管这不会影响某一时刻的指令执行数量(称某一时刻正在执行的指令为飞行指令)。

当然,这种处理器几乎是不可能实现的。例如,IBM Power 5是迄今为止最先进的超标量处理器之一。Power 5每时钟周期最多可发射4条指令,启动执行至多6条指令(对指令类型有严格的限制,如至多2条load-store指令),拥有大量的重命名寄存器(88个定点寄存器和88个浮点寄存器,支持超过200条飞行指令,其中至多可有32条load指令和32条store指令),强大的转移预测器,并且能够动态消除存储器二义性。在讨论完理想处理器的可用指令级并行度之后,我们将研究实际过程中各种功能限制对可用指令级并行度的影响。

为了测量可用的指令级并行度,我们使用标准的MIPS优化编译器对一组程序进行了编译和优化。通过执行和测量,我们得到了这组程序的指令和数据引用的路径记录。这样,在只受限于数据相关的情况下,路径记录中的所有指令都可以通过调度尽可能早地开始执行。同时,使用路径记录可以很容易地实现完美的转移预测和别名分析。在这种机制的支持下,经过调度,指令可以跃过大量与它不存在数据相关性的指令而提早执行,包括跃过转移,因为转移已被准确地预测。

图 3.1 所示为在 6 种 SPEC92 基准测试程序中的平均可用指令级并行度。在这一节中,并行度以平均指令发射速率衡量。需要注意的是,所有的指令均有 1 个时钟周期的时延;而超过 1 个时钟周期的时延将导致每时钟周期发射指令数量的下降。使用的 6 种基准测试程序中有 3 种为浮点密集型基准测试程序 (fpppp, doduc 和 tomcatv), 其余 3 种为定点基准测试程序。在两类浮点基准测试程序中 (fpppp 和 tomcatv) 拥有着极强的指令级并行度,可以由向量计算机或多处理器系统来实现 (由于对代码进行了一些手动转换,因此 fpppp 的结构相当混乱)。doduc 中也存在大量指令级并行度,但与 fpppp 和 tomcatv 不同,其并行度并不出现在简单循环中。程序 li 是一个 LISP 解释器,其中包含大量的近距离相关。

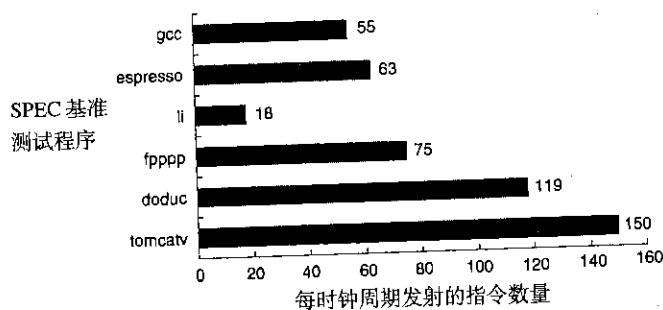


图 3.1 6 种 SPEC 基准测试程序在理想处理器上运行时得到的可用指令级并行度。前 3 种为定点程序,后 3 种为浮点程序。循环密集的浮点程序中蕴涵大量的指令级并行度

在后面几节中,我们将在这个处理器模型上逐步增加一些限制假设,以观察这些假设对指令并行度的影响。之后再讨论实际处理器中的情况。

窗口大小和最大发射数的限制

由于编译阶段的静态分析不可能是完美的,因此,设计一个在转移预测和别名分析上接近理想的处理器需要大量的动态分析工作。当然,实际情况是大多数动态机制也不是完美的,但是,动态分析能够发现大量的指令级并行度,而这一点是编译阶段的静态分析无法实现的。所以,相比静态处理器更容易获得接近理想处理器的并行度。

一个动态调度的推测处理器可以多大程度地接近理想处理器呢?为了回答这个问题,不妨考虑一个理想处理器需要做到的几个方面:

1. 必须可以往前搜索任意距离找到一组可以发射的指令,并正确地预测转移。
2. 可以对所有寄存器进行重命名以避免 WAR 和 WAW 冲突。
3. 判断一组要发射的指令中是否存在数据相关;如是,则进行相应的重命名。
4. 判断要发射的指令中是否存在存储器访问相关,并做出适当的处理。
5. 为就绪指令的发射提供充足的功能单元。

显然,满足这些要求的算法将是非常复杂的。例如,假设所有的指令均为 register-register 指令并且可用寄存器的数量无限,则为了判断要发射的 n 条指令间是否存在寄存器相关,总共需要进行

$$2n - 2 + 2n - 4 + \dots + 2 = 2 \sum_{i=1}^{n-1} i = 2 \frac{(n-1)n}{2} = n^2 - n$$

次比较。因此,为了在 2000 条指令中检测它们之间是否存在相关性,总共需要进行近 400 万次比较!即使只发射 50 条指令,也需要进行 2450 次比较。这个代价显然限制了可一次发射的指令数量。

而在现有的以及短期内可实现的处理器中,相关检测的代价远远不会如此昂贵,因为,我们只需要检测成对的相关性,同时有限的寄存器数量使我们可以采用其他的方法。此外,实际的处理器采用的是按序发射的方式,相关指令由重命名过程花费 1 个时钟周期进行处理。当指令被发射后,相关的检测工作将由保留站或记分板以分布式的方式进行。

为了实现并行执行,需要对指令之间是否存在相关进行检测,被检测指令的集合称为窗口。窗口中的所有指令都必须被保存在处理器中,由于未完成指令必须在所有的已完成指令中查找所需的操作数,因此,处理器每时钟周期需要进行的比较次数等于最大完成速率乘以窗口大小,再乘以每条指令的操作数数量。所以,整个窗口的大小受限于所需的存储容量、能够承受的比较次数以及有限的发射速率,这些因素使得大窗口未必就能发挥更大的作用。需要注意的是,尽管现有的处理器可以支持多达数百条飞行指令,但是由于发射和重命名的能力有限,因此,最大吞吐量有可能受到发射速率的限制。例如,如果指令流中的所有指令都互不相关,且都在 Cache 中命中,则大窗口将永远不会被充满。只有在指令流中存在相关或 Cache 缺失时,大窗口才会比发射率更有价值。

窗口大小直接限制了在给定时钟周期内开始执行的指令数量。在实际情况下,处理器所拥有的功能单元数量有限(比如,没有标量处理器可以在一个时钟周期内处理两条以上的存储器访问),总线和寄存器访问端口的数量也受到限制,这些因素都限制了每时钟周期内可以启动的指令的数量。因此,可以在同一时钟周期内发射、启动执行或提交的指令数量通常要远远低于窗口大小。

显然,多发射处理器的实现受到很多限制:包括每时钟周期发射的指令数量、功能单元及单元延迟、寄存器文件端口、功能单元队列(可能少于单元数)、对转移发射的限制以及对指令提交的限制。所有这些都是影响指令级并行度的因素。在这里,我们并不打算深入研究所有这些因素的影响,而是将主要精力集中在讨论窗口大小的限制,只需知道其他因素也会限制可开发的指令级并行度即可。

图 3.2 说明了窗口大小对并行度的影响。从图中我们可以看到,随着窗口大小的减小,可开发的指令级并行度严重下降。在 2005 年,最先进的处理器的窗口大小在 64~200 之间,但是,实际处理器中的窗口大小与图 3.2 所示的窗口大小并不具有严格的可比性,这主要有两个原因:首先,在我们的例子中所有功能单元的时延均为 1 个时钟周期,而在实际处理器中很多功能单元的时延大于 1 个时钟周期,从而降低了有效窗口大小的可比性;第二,实际处理器中的窗口必须为 Cache 缺失保存存储器引用,而在我们的处理器模型中则假设最优、单时钟周期的 Cache 访问,因此不存在这个问题。

从图 3.2 中我们可以看出,定点程序的指令级并行度要少于浮点程序的指令级并行度。这个结果在我们的意料之中。图 3.2 中并行度的下降轨迹清楚地表明浮点程序中的并行度来自循环级并行。在窗口较小的情况下浮点程序的并行度与定点程序相差无几。这个事实表明在循环体内部存在相关性,而 tomcatv 这类程序的循环迭代之间则很少存在相关性。当窗口较小时,处理器无法判断下一个循环体中的指令能否与当前迭代中的指令一起发射。这个例子很好地解释了先进的编译技术(见附录 G)可以从哪些地方入手来提高指令级并行度,由于先进的编译技术可以发现更多的循环级并行度,并通过调度代码来利用并行的优势,因此即使是在窗口较小的情况下,该方法仍然有效。

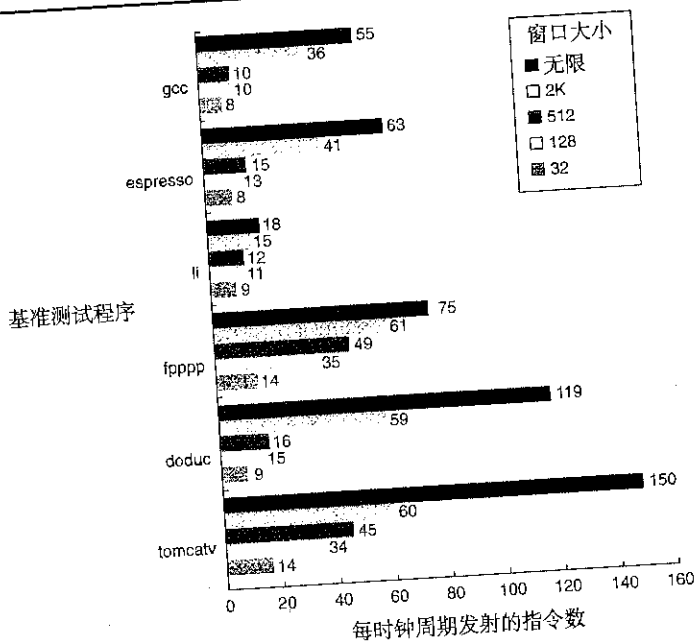


图 3.2 窗口大小对基准测试程序的影响，以每时钟周期发射的指令数目来度量

我们知道太大容量的窗口是不切实际且效率低下的，图 3.2 中的数据表明，与理想情况相比实际的窗口大小会使指令的吞吐量严重下降。因此，在以后的分析中，我们将假设基本窗口大小为 2K，这一大小大约是 2005 年时最大窗口大小的 10 倍，并且假设最大发射能力为每时钟周期 60 条指令，这一速率也是 2005 年时最大发射带宽的 10 倍。我们在后面几节将会看到，对于后面的非理想处理器模型，以上假设不会对可开发的指令级并行度造成限制。

实际转移和跳转预测的影响

理想处理器假设转移总是能够被准确预测：即在程序中的第一条指令执行之前，转移的结果就已经可以被确定了！当然，实际处理器不可能做到这一点。图 3.3 说明了更实际的转移预测带来的影响。图中的数据来自一些不同的转移预测方法，从完美预测到无预测。我们假设有一个独立的预测器用来预测跳转。跳转预测对于最精确的转移预测器来说非常重要，因为对出现频率低的转移而言，转移预测的精确性更为关键。

图中所示转移预测的 5 个等级为

1. 完美：所有的转移和跳转都在执行开始时被准确地预测。
2. 基于 Tournament 的转移预测器：使用相关 2 bit 预测器和不相关 2 bit 预测器，结合选择器。由选择器为每个转移选择最好的预测器。预测缓存包含 213(8K) 个入口，每个入口由三个 2 bit 字段组成，其中两个为预测器，另一个为选择器。相关预测器由转移地址和地址的异或进行索引。非相关预测器是一个标准 2 bit 预测器，由转移地址索引。选择器同样由转移地址索引，它指明了当前情况下应当选择哪个预测器。选择器与标准 2 bit 预测器一样是递增或递减的。该预测器一共使用了 48K bit，其错误率在这 6 种 SPEC92 基准程序中平均为 3%，其策略和容量与 2005 年时最好的预测器大致相当。跳转预测采用 2K 入口的预测器进行，其中一个用于预测返回地址，采用循环缓存的组织形式；另一个

取标准预测器的形式用于预测计算跳转（如 case 语句或 goto 语句）。这些跳转预测器在准确率上几乎接近完美。

3. 有 512 个 2 bit 入口的标准 2 bit 预测器：此外，还假设使用一个有 16 个入口的缓存负责对返回指令进行预测。
4. 基于历史的预测器：静态预测器使用程序的历史文件预测转移是否被选中。
5. 无转移预测：不使用转移预测器，只有跳转仍然被预测。并行在很大程度上被限制在一个基本块之内。

由于我们不考虑由错误的转移预测引起的额外开销，因此，改变转移预测方法只会对基本块之间可开发的并行度产生影响。图 3.4 所示为三种条件转移的实际预测器在我们使用的 SPEC92 基准测试程序子集中的预测准确率。

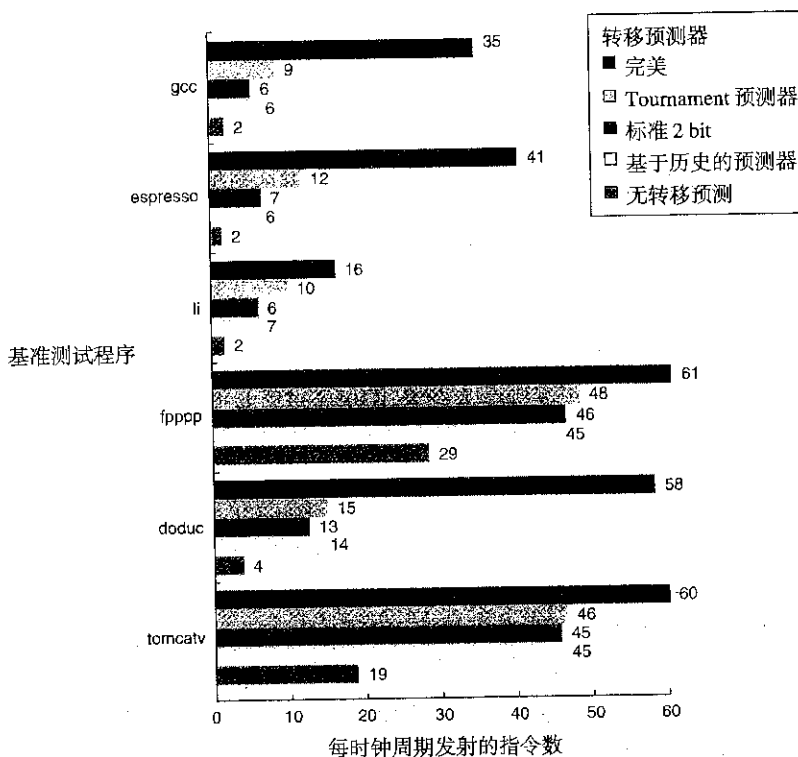


图 3.3 转移预测方法的影响。这幅图依次分析了完美转移预测模型（任意早地准确预测所有转移）；各种动态预测器（选择和 2 bit）；编译阶段的、基于历史的预测器；以及无转移预测对并行度的影响。我们将在后面对这些预测器进行详细介绍。这幅图着重分析了循环级并行度较高的程序（tomcatv 和 fpppp）与循环级并行度较低的程序之间的不同

图 3.3 说明在两种浮点程序中，转移行为要比在其他程序中简单得多，这主要是由于这两种程序的转移较少，且大部分转移的可预测性非常高。这一特点使得我们可以通过实际的转移预测方法开发更高的并行度。而对所有的定点程序以及循环级并行度最低的浮点基准测试程序 doduc 来说，即使是功能最强大的选择预测器也与完美预测器有巨大的差距。与上一小节中获得的数据相似，这些图表说明，为了在定点程序中获得最大并行度，处理器必须选择和执行大量分离的指令。当转移预测不够准确时，预测错误的转移就会成为限制并行性的障碍。

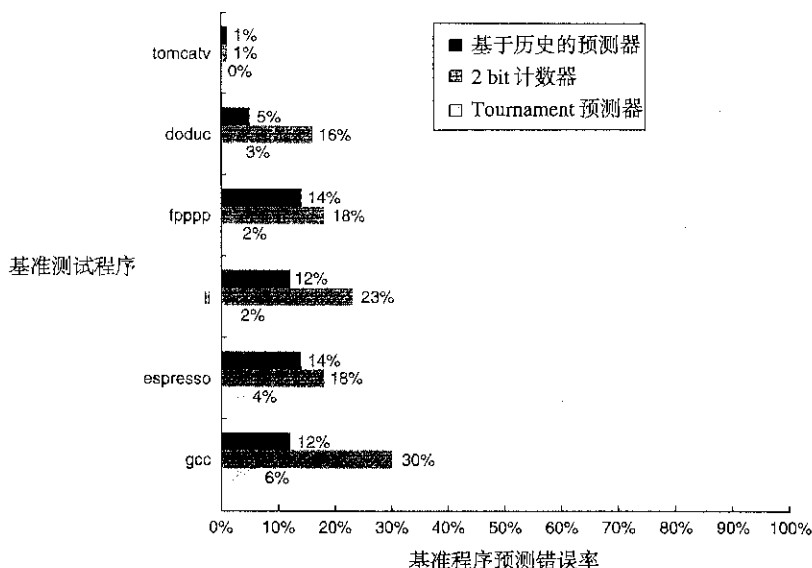


图 3.4 在 SPEC92 子集中条件转移的预测错误率

正如我们所看到的,即使在窗口大小为2K且发射速率为每时钟周期64条指令的情况下,转移预测方法的选择对并行度仍然十分关键。在本节后面的部分中,我们将在已有的对窗口大小和发射速率限定的基础上,补充对转移预测的限制,我们假设使用总共8K个入口、采用两级预测的tournament预测器。这个预测器总共需要150K bits的容量(大约是迄今为止最大预测器的4倍),其性能略强于上面介绍的选择预测器(准确率大约高出0.5%~1%)。同时,我们还假设使用一对上文所述的2K跳转预测器和返回预测器。

有限寄存器的影响

理想处理器模型通过使用无限数量的虚拟寄存器组消除了所有寄存器访问中的名字相关。迄今为止,IBM Power 5是拥有最多虚拟寄存器的处理器:除64个系统结构寄存器外,还补充了88个浮点寄存器和88个定点寄存器。在多线程模式下,所有的这240个寄存器被两个线程共享(见3.5节),并且在单线程模式下所有寄存器对单线程可用。图3.5说明了减少重命名可用寄存器的数量对并行度的影响;在图中,浮点寄存器和定点寄存器的数目随寄存器数量的增长而增长。

图3.5所示的结果可能会令人感到惊讶:读者可能会认为名字相关只会少量减少可用并行度。但需要注意的是,开发并行度需要进行大量的路径分析和推测工作,而这些都需要寄存器为其保存活性变量。图3.5说明在有大量并行度存在的情况下,有限寄存器将会造成非常严重的影响。从图中我们可以看出,有限寄存器对浮点程序的影响较大,而对定点程序的影响则相对较小,这主要是由于窗口大小和转移预测已经严重限制了定点程序的并行度,从而使得有限寄存器的影响不如在浮点程序中突出。此外,应当注意的是,即使增加64个定点寄存器和64个浮点寄存器,使重命名可用的寄存器数量增加到2005年现有处理器的水平,可用并行度的减小仍然会十分明显。

尽管寄存器重命名会对性能产生十分关键的影响,但使用无限数量的寄存器仍然是不现实的。因此,在本节后面的部分中,我们将假设有256个定点寄存器和256个浮点寄存器用于重命名——这一数目已经远远超过了2005年处理器中的实际水平。

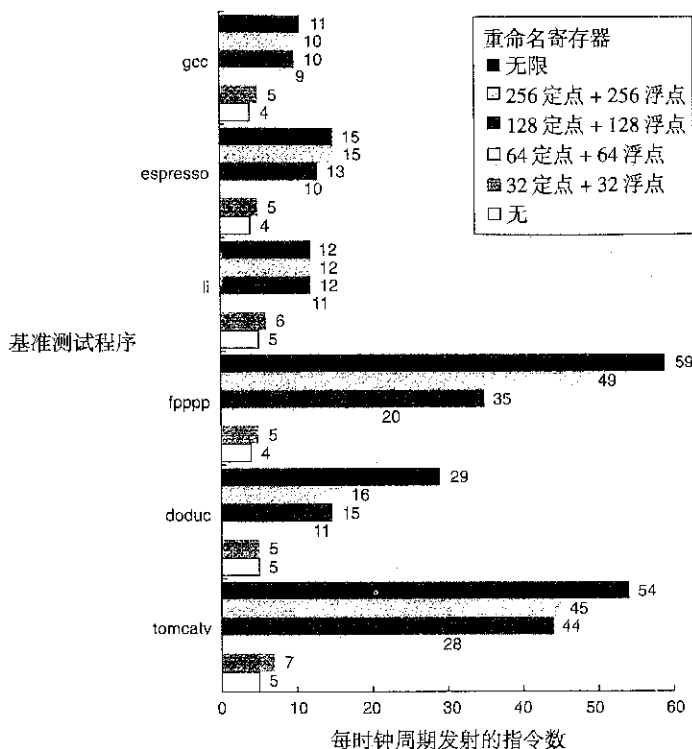


图 3.5 可用并行度随可用重命名寄存器数量的减少而明显下降。浮点寄存器和定点寄存器的数量延 x 轴增加。就是说，“128 定点+128 浮点”意味着总共有 $128 + 128 + 64 = 320$ 个寄存器（其中 128 个用于定点重命名，128 个用于浮点重命名，以及 64 个 MIPS 系统结构中的浮点和定点寄存器）。有限寄存器的影响对浮点程序格外明显，对于这类程序来说，即使只增加 32 个额外定点寄存器和 32 个额外浮点寄存器，也会对并行度产生明显的影响。而对于定点程序来说，即使增加 64 个额外寄存器，效果也不那么明显。使用多于 64 个寄存器需开发大量并行度，对定点程序来说这意味要求有更准确的转移预测

非完美别名分析的影响

优化模型假设可以准确地分析出所有的存储器访问相关，并消除所有的寄存器名字相关。然而，完美的别名分析在实际过程中是不可能实现的：这是由于编译阶段的别名分析不可能是完美的；此外，由于对并行存储器访问的数量没有限制，因此，在运行时可能需要进行无数次比较。图 3.6 所示为完美模型以及其他三种存储器别名分析模型对并行度的影响。这三种模型为

1. **全局/堆栈完美分析模型**：该模型能够准确预测所有的全局和堆栈访问，且假设所有的堆访问都不会发生冲突。该模型是现有产品中基于编译器的分析方法的优化版本。最近的以及正在进行的针对指针的别名分析研究可能会进一步提高堆指针的处理能力。
2. **检测分析模型**：这种模型在编译阶段对访问操作进行检测，以确定它们是否会相互影响。比如，某个访问操作以 R10 作为基址寄存器，偏移量为 20；另一个访问操作也以 R10 为基址寄存器，偏移量为 100。则在 R10 不会被改变的情况下，这两个访问操作不会相互影响。此外，假设基址寄存器指向的地址（如全局域或堆栈域）永远不会是别名。这种分析方法

与许多商用编译器的做法类似,不过新型编译器会做得更好,至少在面向循环的程序中是这样。

3. 无分析: 假设所有的存储器访问都是冲突的。

你可能已经预见到,对于FORTRAN程序来说(FORTRAN程序中没有堆访问),完美预测模型和全局/堆栈完美分析模型没有差别。全局/堆栈完美分析是一种理想化的方法,因为没有编译器能够准确找出所有的相关组合。全局/堆栈完美分析方法的性能大约是检测分析方法的两倍,这个事实表明:要想获得更高的并行度,就必须有复杂编译技术和按需动态分析的支持。在实际情况中,动态调度处理器依赖于动态消除存储器二义性。由于load指令有可能与某条store指令相关,因此,为了消除一条给定load指令的存储器二义性,我们必须要知道所有尚未提交的前序store指令的存储器地址。正如我们在上一章提到的,可以通过存储器地址推测来解决这个问题。

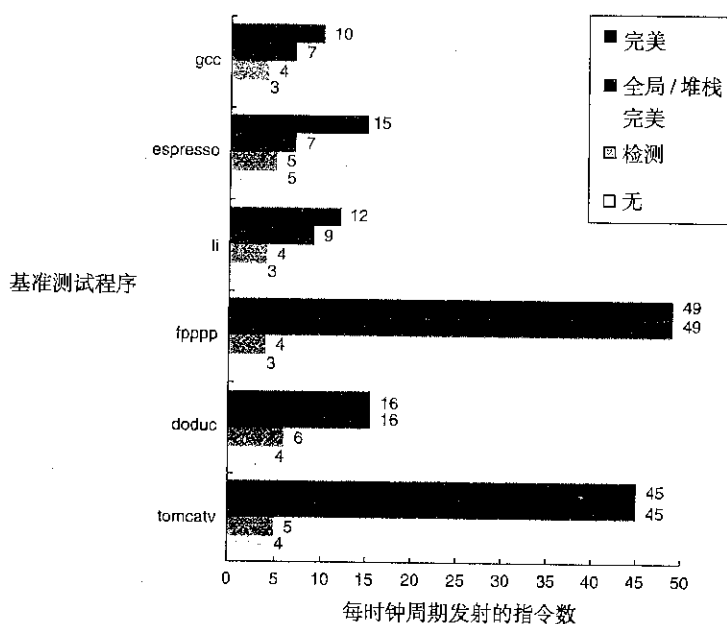


图 3.6 水平不同的别名分析对各个程序的影响。非完美的分析方法会对定点程序产生明显的影响;对于FORTRAN程序来说,全局/堆栈分析是完美的(但不可能实现)

3.3 实际处理器中的指令级并行性限制

在这一节中,我们将讨论有强大硬件支持的处理器性能,该处理器的性能不低于2006年或今后几年将要面世的处理器。具体来说,我们预先设定了以下一组属性:

1. 没有发射限制,每时钟周期至少发射64条指令,这一速度大约是2005年时拥有最大发射带宽的处理器性能的10倍。我们在后面将会提到,实现宽发射处理器所需的高时钟频率、逻辑复杂性和功耗等问题才是限制指令级并行度开发的首要因素。
2. 1K入口的tournament预测器和16入口的返回预测器。这个预测器不逊于2005年时最出色的预测器;预测器并不是主要的瓶颈。
3. 动态地、完美地消除存储器引用的二义性——这一假设虽然看起来野心勃勃,但是却有可能通过小窗口(低发射率和load-store缓存)或存储器相关预测器来实现。

4. 64个附加定点寄存器和64个附加浮点寄存器用于寄存器重命名,这几乎可与IBM Power 5媲美。

图3.7所示为在以上配置中,改变窗口大小时得到的结果。与现有的实现相比,以上这个配置要复杂、昂贵得多,特别是在每时钟周期发射的指令数这一项上,其发射速度超过2005年时拥有最大发射带宽的处理器大约10倍。尽管如此,这个配置毕竟为我们提供了一个目标,并且这个目标在未来是可以实现的。图3.7中所用的数据是非常理想化的,这是由于我们没有对每时钟周期可发射的64条指令进行发射限制。这种能力在未来若干年内是不可能实现的。但遗憾的是,为处理器的发射绑定合理的限制是很困难的,这不仅是因为可能的情况很多,而且因为如果存在发射限制就需要一个精确的指令调度器来对并行度进行判断评估,从而为研究宽发射处理器带来非常昂贵的开销。

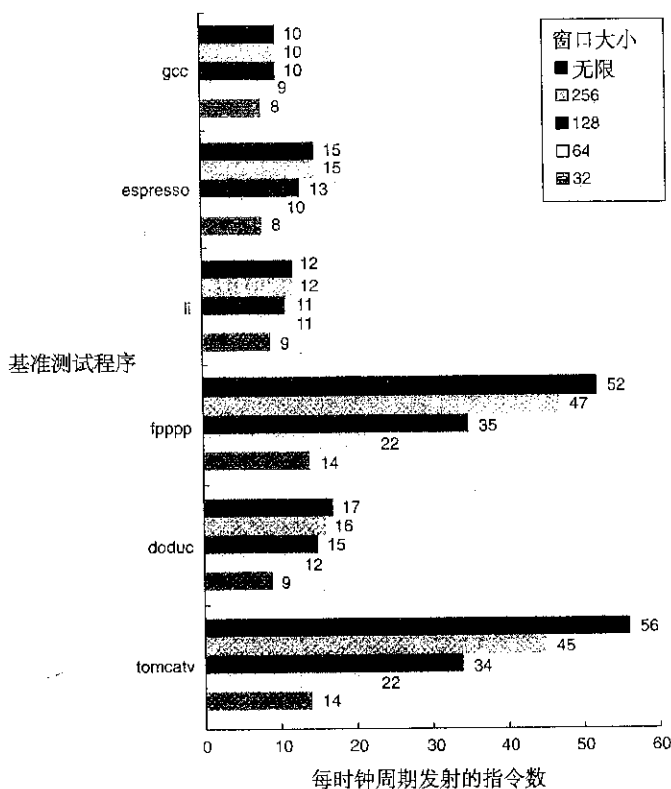


图3.7 在每时钟周期64条指令的发射速率下,一组定点和浮点基准测试程序中的可用并行度同窗口大小的关系。尽管重命名寄存器的数量小于窗口尺寸,但是,由于所有的操作时延均为0,且重命名寄存器的数量等于发射带宽,因此处理器可以在整个窗口内开发并行。在实际情况下,窗口大小和重命名寄存器的数量必须保持平衡,以防止其中之一成为限制发射速率的因素

此外需要注意的是,在对结果进行解释时,Cache缺失和非单元时延的影响都没有被计算在内,而实际情况中这两个因素都会对并行度的开发产生显著的影响。

图3.7中最让人吃惊的现象是,在上述限制条件存在的情况下,窗口大小对定点程序的影响不如对浮点程序的影响大。这个结果反映了这两种程序的一个关键不同。由于在两个浮点程序中循环级并行度较高,因此可开发的指令级并行度也较高;而对定点程序来说,一些其他的因素——如转

移预测、寄存器重命名以及可开发的并行度较少等——都是重要的限制因素。在最近几年里,定点程序性能的重要性日益提高,因此图 3.7 所示的这个现象十分重要。实际上在过去十年中,主要的市场增长——事务处理、Web 服务器等——取决于定点程序的性能而不是浮点程序。我们在下一节中将会看到,一款 2005 年的处理器的实际性能要远远低于图 3.7 所示的水平。

考虑到通过硬件设计提高指令发射速率的难度,设计者们需要解决如何最大限度地利用可用资源的问题。这其中最令人感兴趣的是怎样在简单的、有大容量 Cache 和高时钟频率的处理器与强调指令级并行度但时钟频率较慢且 Cache 容量较小的处理器之间做出权衡。下面的例题很好地阐释了上述挑战:

例题 考虑下面三个假想但比较典型的处理器运行 SPEC 基准测试程序 gcc 时的情况:

1. 简单的 MIPS 双发射静态流水处理器, 时钟频率为 4 GHz, 流水线 CPI 为 0.8。Cache 系统的缺失率为 0.005 每条指令。
2. MIPS 双发射处理器的深度流水版本, 较小的 Cache 和 5 GHz 时钟频率。流水线 CPI 为 1.0, Cache 系统的缺失率平均为 0.0055 每条指令。
3. 窗口大小为 64 个入口的推测执行超标量处理器。其发射率为在该窗口大小下可得到的理想发射率的一半 (使用图 3.7 中的数据)。这种处理器的 Cache 容量最小, 其缺失率为 0.01 每条指令, 但是通过动态调度可以在每次缺失时隐藏 25% 的缺失代价。其时钟频率为 2.5 GHz。

假设所有的存储器访问时间 (决定缺失代价) 为 50 ns。试判断这三种处理器的相对性能。

解答: 首先, 使用缺失率和缺失代价计算在以上这三种处理器中 Cache 缺失对 CPI 的影响。使用下面的公式:

$$\text{Cache CPI} = \text{缺失率} \times \text{缺失代价}$$

需要为以上三种处理器分别计算缺失代价:

$$\text{缺失代价} = \frac{\text{内存访问时间}}{\text{时钟周期}}$$

三种处理器的时钟周期分别为 5 ps, 200 ps 和 400 ps。因此缺失代价为

$$\text{缺失代价}_1 = \frac{50 \text{ ns}}{250 \text{ ps}} = 200 \text{ 个周期}$$

$$\text{缺失代价}_2 = \frac{50 \text{ ns}}{200 \text{ ps}} = 250 \text{ 个周期}$$

$$\text{缺失代价}_3 = \frac{0.75 \times 50 \text{ ns}}{400 \text{ ps}} = 94 \text{ 个周期}$$

把结果代入 Cache CPI 的公式:

$$\text{Cache CPI}_1 = 0.005 \times 200 = 1.0$$

$$\text{Cache CPI}_2 = 0.0055 \times 250 = 1.4$$

$$\text{Cache CPI}_3 = 0.01 \times 94 = 0.94$$

处理器 3 的 CPI 流水线为

$$\text{流水线 CPI}_3 = \frac{1}{\text{发射率}} = \frac{1}{9 \times 0.5} = \frac{1}{4.5} = 0.22$$

将流水线 CPI 与 Cache CPI 相加得到各个处理器的 CPI:

$$CPI_1 = 0.8 + 1.0 = 1.8$$

$$CPI_2 = 1.0 + 1.4 = 2.4$$

$$CPI_3 = 0.22 + 0.94 = 1.16$$

由于三种处理器的系统结构都是相同的,因此我们可以通过比较指令的执行速率(单位为百万条指令/秒, MIPS)来衡量三种处理器的相对性能:

$$\text{指令执行速率} = \frac{CR}{CPI}$$

$$\text{指令执行速率}_1 = \frac{4000 \text{ MHz}}{1.8} = 2222 \text{ MIPS}$$

$$\text{指令执行速率}_2 = \frac{5000 \text{ MHz}}{2.4} = 2083 \text{ MIPS}$$

$$\text{指令执行速率}_3 = \frac{2500 \text{ MHz}}{1.16} = 2155 \text{ MIPS}$$

可见在这三种处理器中,简单双发射静态超标量处理器的相对性能最好。在实际情况中,性能取决于 CPI 和时钟频率。

克服研究模型的限制

与所有其他研究一样,我们在这一节中所做的研究也有它自己的局限。主要可分为两类:一类是即使在完美推测执行处理器模型中也存在的限制,另一类是实际处理器模型带来的限制。当然,所有来自第一类的限制都会作用于第二类。完美模型中存在的主要限制有:

1. 访问存储器的 WAW 和 WAR 冒险: 我们在这一节所做的研究中通过寄存器重命名消除了 WAW 和 WAR 冒险,但是并没有涉及存储器访问的 WAW 和 WAR 冒险。尽管表面上看起来这类问题似乎很少发生(特别是 WAW 冒险),但是在堆栈页面分配时这些问题还是会出现。某个操作重用前一个操作使用过的堆栈空间,就会引起 WAW 和 WAR 冒险,从而带来不必要的限制。Austin 和 Sohi [1992]研究了这个问题。
2. 多余的相关: 在寄存器数量无限的情况下,所有访问寄存器的名字相关都会被消除,只有真寄存器数据相关被保留下来。但是递归或代码生成的约定仍然会产生一些多余的真数据相关。比如简单 do 循环中的控制变量的相关: 由于控制变量在每次循环时都会递增,因此循环中至少存在一个相关。我们在附录 G 中将会看到,循环展开以及功能强大的代数优化可以消除这种相关计算。Wall 对这类优化技术进行了一定的研究,更积极地使用这种技术可以带来更高的指令级并行度。此外,一些代码生成约定也会引起不必要的相关,这种问题在使用返回地址寄存器和堆指针寄存器时(在调用/返回队列中递增递减)格外突出。Wall 的研究消除了返回地址寄存器的影响,但是在链接约定中对堆指针的使用仍然会引起不必要的相关。Postiff 等[1999]探讨了这一问题。
3. 克服数据流的限制: 在准确率足够高的情况下,值预测技术可以克服数据流的限制。但是到目前为止,在关于这一主题的 50 多篇论文中,没有一篇能通过实际的预测方法来明显提高指令级并行度。显然,完美的数据值预测可以带来无限的有效并行度,因为所有指令的所有值都可以被准确地提前预测。

对非完美处理器, 研究人员提出了多种思想来开发更多的指令级并行, 如多路推测。Lam 和 Wilson[1992]探讨了 this 思想, 本节中也涉及到这方面的内容。多路推测可以降低错误恢复的代价, 同时可以开发更多的指令级并行度。由于所需的硬件资源随转移个数呈指数增长, 因此多路推测只在转移数目有限的情况下才有意义。Wall[1993]提供了在最多八路转移上进行双向推测的数据。考虑到双向推测的代价, 并且已知其中一个转移方向会被丢弃(由于该转移方向还会包括其他的转移, 因此会使无意义的计算量增加), 因此, 商用设计并没有在双向推测上投入更多的精力, 而是为了提高预测准确率而补充了额外的硬件。

有一点需要说明的是, 本节中讨论的这些限制理论上并不是无法克服的! 实际上, 这些限制是我们在开发指令级并行的过程中所遇到的巨大障碍。这些限制——不管它是窗口大小、别名分析还是转移预测——都是设计者和研究人员们所必须面对的挑战。正如我们在 3.6 节中将要提到的, 指令级并行性限制的延伸以及实现宽发射的代价才是限制开发指令级并行的主要因素。

3.4 相关问题: 硬件推测和软件推测

在“相关问题”中, 我们讨论的主题将涉及到其他章的一些内容。后面几章都将包含“相关问题”一节。

上一章介绍的硬件推测方法和附录 G 中介绍的软件方法为我们开发指令级并行度提供了不同的选择。怎样在硬件方法和软件方法之间做出权衡? 这两种方法的限制都有哪些? 我们将在下面列举相关内容:

- 为了进行推测, 我们必须消除存储器访问的二义性。对于包含指针的定点程序来说, 要想在编译阶段完成这项工作是非常困难的。在硬件方法中, 动态的、运行时的存储器地址二义性消除是通过 Tomasulo 算法实现的。这使得我们可以在运行时将 load 指令移动到 store 指令之前。对存储器访问的推测可以使我们克服保守的编译器带来的限制, 但是对推测技术的盲目使用可能会引起昂贵的恢复开销, 从而使推测技术的优势消耗殆尽。
- 当控制流不可预测且基于硬件的转移预测优于编译阶段的软件转移预测时, 基于硬件推测的工作效率要比基于软件的方法更加出色。很多定点程序都具有这样的特点。比如, 对四种主要的定点 SPEC92 程序来说, 一款出色的静态预测器的预测错误率大约为 16%, 而硬件预测器的错误率则不到 10%。之所以会有如此明显的差距, 是因为当预测错误时, 推测指令会大大降低计算速度。也正是由于这个差距, 即使是静态调度的处理器一般也要在内部包含动态转移预测器。
- 基于硬件的推测能够维护完整、精确的异常模型, 即使对推测指令来说也是这样。为了实现这个目标, 最近出现的基于软件的方法也补充了一些特殊的支持。
- 基于硬件的推测不需要补充代码或代码记录, 而在软件推测方法中这些条件都是必需的。
- 与纯硬件方法相比, 基于编译器的方法可调度的代码量更大, 因此其在代码调度方面也更加出色。
- 对于同一系统结构的不同实现, 动态调度的硬件推测能够在不改变代码序列的情况下获得出色的性能。尽管很难对这一优势进行量化, 但从长远来看, 硬件推测的这一特点无疑是十分重要的。有趣的是, 这正是 IBM 360/91 的设计动机之一。而另一方面, 最新的并行系统结构, 如 IA-64 等, 已经为减小代码对硬件的内在依赖增加了新的灵活性。

对于基于硬件的推测方法来说,所需的硬件资源及复杂性是其所面临的一个主要问题。在对硬件推测和软件推测进行比较时,必须对硬件方法的这类开销做出评价,并将其与软件方法中编译器的复杂性以及基于该编译器的处理器的简化意义进行对比。我们将在结论部分继续讨论这个问题。

一些设计者尝试将软件方法和硬件方法结合起来,使它们各自发挥自己的优势。这种结合会在两种方法之间产生一些有趣的、不确定的相互作用。比如,把条件移动指令同寄存器重命名结合在一起时产生的副作用:由于之前已经在指令流水中进行了重命名,因此被撤销的条件移动仍然会将结果复制到目标寄存器。这种相互作用会使设计和验证过程变得更加复杂,同时也会降低性能。

3.5 多线程:使用指令级并行支持线程级并行的开发

尽管对程序员透明是指令级并行的一个主要优势,但是正如我们前面所提到的,对于某些程序来说,开发指令级并行是非常困难的。而且,在一些应用的高层中可能蕴藏着大量的并行度,而开发指令级并行度的方法却对这种高层并行度无能为力。比如,在联机事务处理系统中,查询操作和更新操作之间存在大量的天然并行度。由于互不相关,因此大部分这类查询操作和更新操作都可以并行处理。当然,在许多科学应用中也存在这种天然并行度,这是由于科学应用是自然并行结构的三维模型,这种结构可以在模拟中开发。

由于逻辑结构为可独立执行的线程,因此我们称上面提到的这种高层并行度为线程级并行(TLP)。线程是指可以独立执行的进程,它拥有自己的指令和数据。一个线程可以是一个包含有很多进程的并行程序的一部分,也可以是一个独立的程序。线程具备执行的所有条件(指令、数据、PC、寄存器状态等)。与开发循环或直线型代码中的并行操作的指令级并行不同,线程级并行的目标是开发多个执行线程之间内在的并行性。

线程级并行是一种可替代指令级并行的重要方法,这主要是因为开发线程级并行所需要的成本要比指令级并行低得多。线程级并行在许多重要应用中天然存在,比如大多数服务器应用。对于定制的软件来说,表达程序内在的并行性是很容易的,比如在一些嵌入式应用中的情况。但是对于那些大型而且完整的应用来说,如果在开发过程中没有考虑到并行性的问题,那么通过重写软件来开发并行度将会花费昂贵的代价。第4章将讨论多处理器,以及通过多处理器支持线程级并行的问题。

线程级并行同指令级并行开发的分别是程序并行结构中的两个不同方面。一个很自然的问题是,面向指令级并行设计的处理器能否开发线程级并行呢?之所以会有这样的问题,是因为我们发现:在开发指令级并行的过程中,由于代码中存在相关或停顿,因此在数据路径上经常会有一些空闲的功能单元。线程间的并行能否作为不相关指令源,在停顿期间使处理器始终保持工作状态呢?在缺乏指令级并行度的情况下,线程级并行能否将空闲的功能单元利用起来呢?

多线程使多个线程以重叠的方式共享一个处理器中的功能单元。为了支持这种共享,处理器必须为每个线程保存指令状态。比如,每个指令都需要有一个独立的寄存器文件复本、一个独立的PC以及一个独立的页表。存储器自身可以通过虚拟存储器机制实现共享,这种机制已经为多道程序系统提供了支持。此外,硬件还必须对不同线程之间的快速切换提供支持,特别是线程切换的效率应当比进程切换高得多,后者通常需要花费数百甚至数千个处理器周期。

多线程主要有两种方法。细粒度多线程能够在指令之间进行线程切换,从而使多个线程交替执行。这种交替通常采用轮流的方式,跳过某一时刻的所有停顿线程。为了实现细粒度多线程,处理器必须具备在任一时钟周期切换线程的能力。细粒度多线程的一个关键优点在于:通过在某个线程停顿时执行其他线程中的指令,细粒度多线程可以隐藏由停顿引起的吞吐量损失。而它的主要缺点

在于降低了每个线程的执行速度,这是由于没有停顿的、已经准备好执行的指令可能会被来自其他线程的指令阻塞。

粗粒度多线程是作为细粒度多线程的替代方法而设计的。粗粒度多线程只在发生代价较高的停顿时才切换线程,比如二级 Cache 缺失。由于在粗粒度多线程方法中,只有遭遇高代价停顿时才会发射其他线程中的指令,因此同细粒度多线程相比,粗粒度多线程降低了线程切换的代价,很大程度上避免了降低处理器速度的问题。

但粗粒度多线程也有自己的问题,即克服吞吐量损失的能力有限,特别是在停顿时间较短时这个问题会更加突出。这是由粗粒度多线程的流水线启动开销引起的。由于粗粒度多线程的 CPU 从每个单独的线程发射指令,因此当停顿发生时,流水线必须被清空或暂停。而停顿后开始执行的新线程必须先填满流水线才能完成指令。由于启动开销的影响,粗粒度多线程多用于减小长停顿的代价,在这种情况下,同较长的停顿时间相比,重新填满流水线的代价通常是可以忽略不计的。

在下一小节中我们将探讨细粒度多线程的各个改进版本,这些技术使得超标量处理器能够完整、有效地开发指令级并行和多线程。在第4章中,在讨论多线程和多处理器在单个芯片上的整合时,我们还会继续研究多线程的问题。

同时多线程:将线程级并行转换为指令级并行

同时多线程(SMT)是多线程的一个改进版本,它使用多发射和动态调度处理器在开发线程级并行的同时开发指令级并行。同时多线程产生的主要原因在于,现代多发射处理器的功能单元中通常含有大量的并行度,而单个线程无法有效地利用这种并行度。此外,通过寄存器重命名和动态调度,来自各个独立线程的多条指令可以被同时发射,而不考虑指令间的相关性;相关由动态调度负责处理。

图 3.8 从概念上阐述了以下几种配置的处理器在开发超标量资源能力上的不同:

- 不支持多线程的超标量处理器
- 支持粗粒度多线程的超标量处理器
- 支持细粒度多线程的超标量处理器
- 支持同时多线程的超标量处理器

在不支持多线程的超标量处理器中,发射槽的利用率受到缺乏指令级并行度的限制,我们在前面已经讨论过这个问题。此外,像指令 Cache 缺失这类主要的停顿有可能使整个处理器陷入空闲状态。

支持粗粒度的超标量处理器可以在一个线程停顿时,通过切换使另一个线程利用处理器资源,这样做可以部分隐藏长停顿的代价。尽管这样可以减小完全空闲的时钟周期数,但是在每个时钟周期内部,指令级并行的限制仍然会引起空闲。此外,在粗粒度超标量处理器中,由于只在停顿时才进行线程切换,且新线程的启动需要一段时间,因此,仍然有可能出现完全空闲的时钟周期。

在支持细粒度的超标量处理器中,线程的交替消除了完全空的发射槽。但是由于在给定的时钟周期内只允许一个线程发射指令,因此在指令级并行的限制下,每个时钟周期内部还是会有空闲的发射槽。

在同时多线程中,所有的发射槽在一个时钟周期内被多个线程共享,线程级并行和指令级并行被同时开发。理想情况下,发射槽的利用率仅受限于多个线程对资源的需求同可用资源之间的不平衡。在实际情况中,其他的因素——包括活跃线程的数量、有限缓存的限制、从多线程中取到足够指令的能力,以及对一个或多个线程允许发射的指令组合的约束——也是限制发射槽使用率的因素。

尽管图 3.8 大大简化了这些处理器的操作,但它还是很好地阐释了一般多线程,特别是同时多线程中潜在的性能优势。

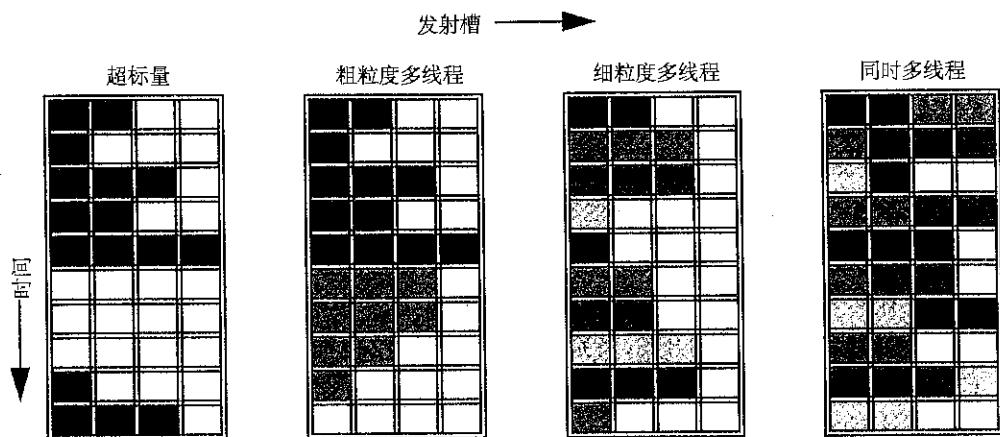


图 3.8 上述四种方法对超标量处理器发射槽的使用率。水平方向表示每个时钟周期内的指令发射能力。垂直方向表示时钟周期序列。空格(白色)表示时钟周期内未使用的指令发射槽。灰色和黑色分别对应多线程处理器中的四个不同线程。在不支持多线程的超标量处理器中,黑色也用来表示被占用的发射槽。我们将在下一章中讨论的 Sun T1(aka Niagara)处理器是一款细粒度多线程系统结构的处理器

我们在前面曾经提到,同时多线程假设动态调度处理器已经拥有了通过多线程开发线程级并行的硬件支持。特别是,动态调度超标量处理器拥有大量的虚拟寄存器,可以为各个独立的线程保存寄存器组(假设每个线程都有独立的重命名表)。由于重命名寄存器提供了唯一的寄存器标识,因此来自多个线程的指令可以混杂在数据路径中,而不引起多个线程间源和目标的混淆。

通过为每个线程设置重命名表、保留指令各自的 PC 值、为来自多个线程的指令的提交提供支持,多线程完全可以在乱序执行处理器的基础上实现。

由于来自独立线程的指令需要以独立的方式提交,因此对指令提交的处理会比较复杂。这可以通过为每个线程保留独立的重排序缓存的方法来实现。

同时多线程的设计挑战

动态调度超标量处理器通常都采用深度流水,因此,使用粗粒度方法实现的同时多线程不可能获得出色的性能。由于同时多线程只在细粒度的实现方式下才有意义,因此,我们必须需要考虑细粒度调度对单个线程性能的影响。我们可以通过使用一个优先线程,以单线程性能的少量牺牲换取多线程的整体性能,从而将这种影响减小到最小。

初看起来,优先线程的方法似乎既不会牺牲吞吐量,也不会降低单线程的性能。但遗憾的是,当优先线程停顿时,处理器确实会损失掉部分吞吐量。这是由于流水线中的指令不可能总是来自多个线程的混合,这会增大空发射槽和停顿的可能性。足够多的独立线程可以隐藏线程停顿的代价,从而使吞吐量最大化。

遗憾的是,将多个线程混合在一起将不可避免地影响单个线程的执行时间。在取指令的过程中也存在类似的问题。为了最大限度地维持单个线程的性能,我们应当尽可能早地为单个线程取指,并在转移预测错误和预取缓存缺失时清空取指单元。但是这种做法必定会限制其他线程可调度的指令数,降低吞吐量。所有的多线程处理器都必须在这个矛盾中做出权衡。

那么怎样才能平衡单线程性能同多线程性能之间的矛盾呢?实际上这个问题并不像看上去那么困难,至少对于当前超标量处理器的后端设计来说是这样。特别是对于每时钟周期发射4~8条指令的现代处理器来说,通常只需要少量的活跃线程就足够了,而对优先线程数量的要求甚至会更少。只要有可能,处理器就会运行优先线程。从预取指令开始:无论何时,只要优先线程的预取缓存未被填满,处理器就为它优先取指。只有当优先线程的预取缓存被充满时,取指单元才会为别的线程取指令。但是需要注意的是,如果有两个优先线程,那我们就要同时预取两个指令流,这会增加取指单元和指令Cache的复杂度。与此类似,指令发射单元也会首先照顾优先线程,只在优先线程停顿或是无法发射的情况下才会考虑其他的线程。

同时多线程处理器的设计还面临一些其他的挑战,包括:

- 为了保存多个上下文信息,寄存器文件会格外庞大
- 不能影响每个时钟周期的开销,特别是在关键步骤上,比如在指令发射步骤中需要考虑更多的候选指令,在指令完成步骤中需要对提交的指令进行选择
- 必须保证同时执行多个线程引起的Cache和TLB的冲突不会使性能显著下降

在讨论这些问题时,有两个观点非常重要。首先,在大多数情况下,由多线程引起的开销通常是很小的,简单的选择已经足够应付;第二,当前超标量处理器的效率相当低,还有很大的上升空间,在这种情况下即使付出一些代价也是值得的。

IBM的Power 5的流水线在Power 4的基础上增加了对同时多线程的支持。在增加了同时多线程后设计人员发现,为了将细粒度线程交互对性能造成的消极影响减到最小,他们不得不增加处理器中结构体的数量。具体包括:

- 增加一级指令Cache和指令地址转换缓存之间的相关性
- 增加每个线程的load和store队列
- 增加二级和三级Cache的容量
- 增加独立的指令预取和缓存
- 将虚拟寄存器的数量由152个增加到240个
- 增加发射队列的容量

由于同时多线程是在多发发射超标量处理器上开发线程级并行度,因此它通常应用于以服务器市场为目标的高端处理器中。此外,有些模式可能会对多线程进行约束,以最大化单个线程的性能。

同时多线程的潜在性能优势

一个关键的问题是,同时多线程到底能使性能有多大提高?当研究人员在2000年到2001年之间讨论这个问题时,他们假设动态超标量处理器的发射带宽将在未来5年内被拓宽,达到每时钟周期发射6~8条的水平,同时还假设处理器支持推测动态调度、大量的并行load和store、大容量的Cache以及同时支持多个上下文且其中4~8个可以并行取指。但是由于种种原因,在目前以及未来的一段时间内,恐怕没有哪一款处理器能够拥有这样的能力,我们在下一节会明确地看到这一点。

研究人员们过分乐观的假设所造成的结果就是,尽管他们的研究表明在多进程负载中,同时多线程可以使性能提高两倍以上,但是这个结果是不切实际的。现有的同时多线程的实现通常只支持两个上下文,其中只有一个可以取指,并且指令发射能力也同当初设想的有较大差距。因此,同时多线程对性能起到的实际作用也比当初的研究结果逊色得多。

比如HP-Compaq服务器采用的Pentium 4 Extreme,在运行SPECintRate基准测试程序时,同时多线程使性能提升了1.01倍,运行SPECfpRate时性能提升了1.07倍。Tuck和Tullsen [2003]的

研究表明, 当将 26 个 SPEC 基准测试程序中的任意两种配对运行时 (即 262 次运行, 如果每个基准测试程序也与自己配对的话), 加速比在 0.90~1.58 间浮动, 平均加速比为 1.20。在运行 SPLASH 并行基准测试程序时, 多线程的加速比在 1.02~1.67 之间, 平均为 1.22。

IBM Power 5 是到 2005 年为止同时多线程的最强大实现, 它为了支持同时多线程做了大量扩展, 我们在上一小节中已经对此做了介绍。考虑下面这两种情况: 一种为在 Power 5 的同时多线程模式下由一个处理器运行某个应用的两个副本; 另一种为在单线程模式的 Power 5 中运行该应用的两个副本, 每个内核一个进程。对这两种情况下性能的直接比较表明, 对大多数基准测试程序来说, 加速比在 0.89 (性能损失) 到 1.41 之间。对于大多数应用来说, 同时多线程或多或少都使性能得到了提升; 而对浮点密集型应用来说, 由于 Cache 冲突的情况最为严重, 因此从同时多线程中获益最少。

图 3.9 为在 8 处理器的 Power 5 上运行 SPECRate2000 时, 同时多线程模式对单线程模式的加速比。SPECintRate 的平均加速比为 1.23, SPECfpRate 的加速比平均为 1.16。需要注意的是, 一些浮点基准测试程序在同时多线程模式下的性能稍有下降, 最低的加速比为 0.93。

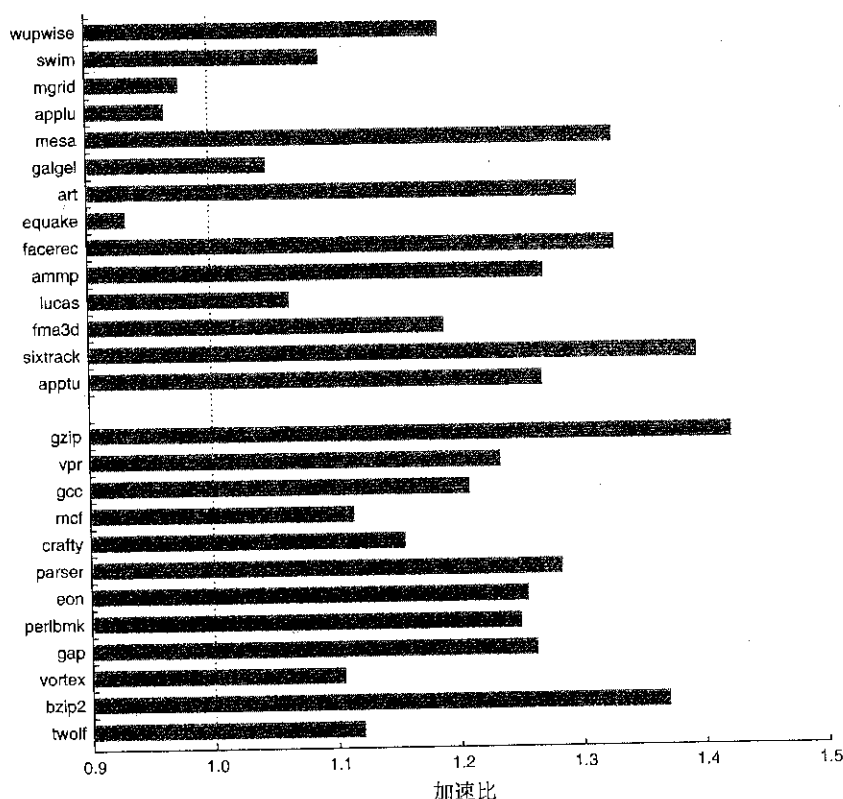


图 3.9 同时多线程和单线程模式下, 8 处理器 IBM eServer p5 575 的性能比较。y 轴的起点为 0.9, 意味着 0.1 的性能损失。在每个 Power 5 核心中只有一个处理器是活跃的, 这样做可以降低存储系统的互相干扰, 从而改进同时多线程的效果。同时多线程模式下的结果是通过运行 16 个用户线程获得的, 而单线程模式只运行 8 个线程; 通过使每个处理器运行一个线程, Power 5 由操作系统切换到单线程模式。图中所示结果由 IBM 的 John McCalpin 收集。我们从这些数据中可以看出, SPECfpRate 的标准偏差要比 SPECintRate 的高 (0.13 对 0.07), 这表明同时多线程对浮点程序的影响变数较大

这个结果清楚地表明了强大的推测处理器中支持同时多线程所带来的好处。但是考虑到代价以及性能递减的问题,与实现更宽的超标量处理器和更强大的同时多线程相比,大多数设计者还是更倾向于在单独的 CPU 晶片上实现多个核心,同时辅以轻量级的多发射和多线程的支持。我们将在下一章中继续讨论这个问题。

3.6 综合:高级多发射处理器的性能和效率

在这一节中,我们将讨论当今一些多发射处理器的特性,研究它们对硅晶体、晶体管以及能量的利用率。之后我们将讨论超标量处理器的实际限制以及高性能多处理器的未来。

图 3.10 所示为当前最新的四种高性能处理器的特性。它们在组织结构、发射率、功能单元的能力、时钟频率、晶片大小、晶体管数量以及功耗方面都有很大的不同。如图 3.11 和图 3.12 所示,在性能方面这四种处理器中并没有哪一款特别突出。在使用 SPECfp 进行测试时,Itanium 2 和 Power 5 性能相似,都远远超过了 Athlon 和 Pentium 4 在运行这类基准测试程序时的表现。而在使用 SPECint 进行测试时,AMD Athlon 的性能最为出色,其次依次是 Pentium 4, Itanium 2 和 Power 5。

处理器	微系统结构	取指/发射/ 执行	功能单元	时钟频率 (GHz)	晶体管和 晶片大小	功耗
Intel Pentium 4 Extreme	推测动态调度 深度流水 同时多线程	3/3/4	7 定点 1 浮点	3.8	125M 122 mm ²	115 W
AMD Athlon 64 FX-57	推测动态调度	3/3/4	6 定点 3 浮点	2.8	114M 115 mm ²	104 W
IBM Power 5 1 processor	推测动态调度 同时多线程 每芯片两个 CPU 核心	8/4/8	6 定点 2 浮点	1.9	200M 300 mm ² (估算)	80 W (估算)
Intel Itanium 2	EPIC 风格 静态调度为主	6/5/11	9 定点 2 浮点	1.6	592M 423 mm ²	130 W

图 3.10 当前四种多发射处理器的特性。Power 5 包括两个 CPU 核心,但是我们在本章中只研究一个核心的性能。双核 Power 5 的晶体管数量、面积以及功耗分别为 276M, 389 mm² 和 125 W, 图中单核 Power 5 的数据是以此为基础估算出来的。Itanium 2 中较大的晶片大小和晶体管数量是由 9 MB 的片上三级 Cache 造成的。AMD Opteron 和 Athlon 拥有相同的核心微系统结构。Athlon 为桌面市场设计,不支持多线程;Opteron 为服务器市场设计,支持多线程。这与 Intel 产品线中 Pentium 和 Xeon 的关系相似

和总体性能同等重要的问题是对硅晶体面积和功耗的利用率。正如我们在第 1 章中讨论的,功耗已经成为了限制当代处理器的首要因素。图 3.13 通过将这四款处理器运行 SPECint 和 SPECfp 时的性能同晶体管的数量、硅晶体面积以及功耗进行对比,给出了它们的效率。图 3.13 所示的效率结果与性能结果构成了有趣的对比。除了运行 SPECfp 时对功耗的利用较好外,Itanium 2 的各种效率在所有的浮点和定点基准测试程序中都是最差的。Athlon 和 Pentium 4 在对晶体管和硅晶体面积的利用上效率较高,而 IBM Power 5 在运行 SPECfp 时的功耗利用率是最高的,运行 SPECint 时的功耗利用率几乎和效率最高的 AMD Athlon 持平。事实表明,如果综合考虑各个方面的利用率,这四

款处理器中没有哪一款能够占有压倒性的优势,也就是说在目前的水平下,这些方法当中没有哪一种是可以用来高效开发指令级并行的捷径。

下面让我们来研究为什么会出现这种情况。

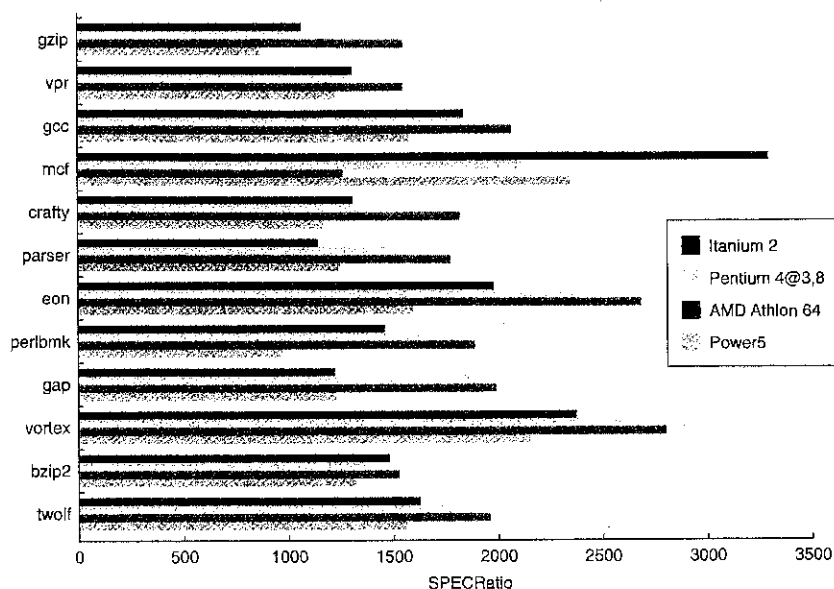


图 3.11 使用 SPECint2000 基准测试程序进行测试时, 图 3.10 中所示的四种多发射处理器的性能比较

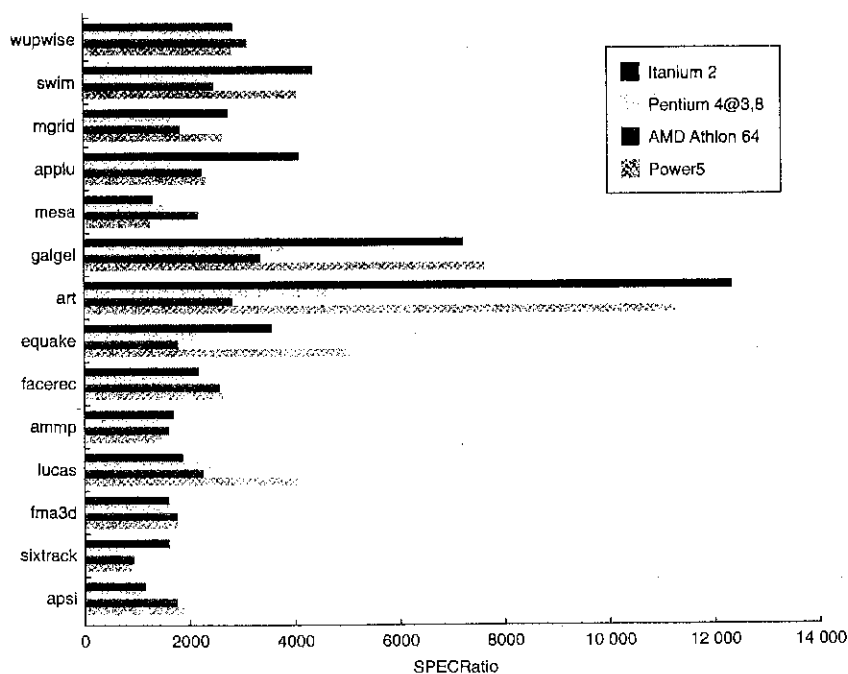


图 3.12 使用 SPECfp2000 基准测试程序进行测试时, 图 3.10 中所示的四种多发射处理器的性能比较

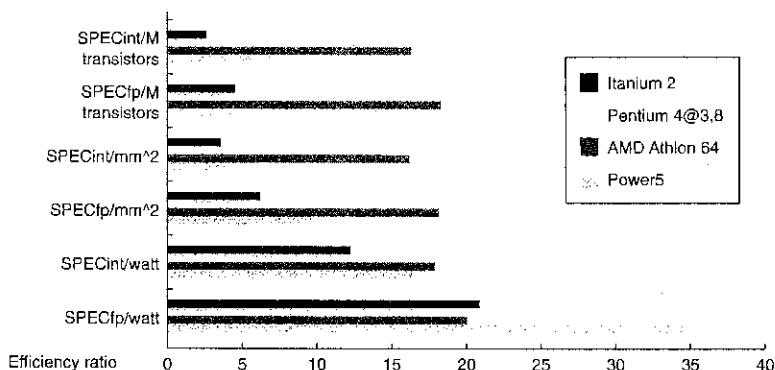


图 3.13 四种多发射处理器效率的比较。在 Power 5 中, 一个晶片内包含两个处理器核心, 我们估算单个核心的值分别为: 功耗 = 80 W, 面积 = 290 mm², 晶体管数量 = 200M

是什么限制了多发射处理器

我们曾经在 3.1 节和 3.3 节中讨论了限制开发指令级并行的主要障碍。比如, 为了使指令发射速度增长一倍, 即从每时钟周期 3~6 条指令增长到每时钟周期 6~12 条指令, 处理器需要每时钟周期发射 3~4 条数据存储器访问、处理 2~3 条转移、对 20 多个寄存器进行重命名和访问、取 12~24 条指令。这些能力的实现相当复杂, 这意味着要以最大时钟频率为牺牲。例如, 在图 3.10 中, 拥有最大发射带宽的处理器是 Itanium 2, 而 Itanium 2 的时钟频率也是最慢的, 同时 Itanium 2 对功耗的利用率也是最低的。

目前已经逐渐形成的一个共识是, 功耗是限制当代处理器的首要因素。功耗是静态功耗和动态功耗的函数。静态功耗随晶体管的数量成比例增长 (而不管晶体管是否切换), 而动态功耗与切换晶体管的次数和切换速率的积成比例关系。尽管静态功耗也是设计过程中必须要考虑的问题, 但是动态功耗通常才是能源的主要消费者。一款处理器要想同时获得较低 CPI 和较快的时钟频率, 那么它必须能够快速地在大量的晶体管之间进行切换, 而功耗的消费为这二者的积。

当然, 包括多核和多线程在内的大多数提高性能的技术都会增加功耗。关键的问题是这些技术是否是能源有效的, 即功耗增加的速度是否比性能提升的速度要快。遗憾的是, 在这方面, 目前用来提升多发射处理器性能的技术都是低效的, 这主要是由于以下两种原因。

首先, 多发射会增加一些逻辑开销, 而这类开销的增长速度要快于发射率的增长速度。这类逻辑负责指令发射分析, 包括相关检测、寄存器重命名以及各种类似的功能。它们综合作用的结果就是, 由于开销的影响, 在不降低功率的情况下, CPI 的降低会使每瓦特得到的性能也相应降低。

第二且更为重要的一点是发射速率的峰值与持续性能之间差距的拉大。由于晶体管切换的次数同发射速率峰值成比例关系, 而性能同持续速率成比例关系, 因此发射率峰值同持续性能之间距离的拉大, 会增加每性能单元所消耗的能源。遗憾的是, 这种差距的拉大是内在的也是必然的, 这是由我们在 3.2 节和 3.3 节中讨论的问题所引起的。比如, 如果我们要维持每时钟周期发射 4 条指令的速度, 那么我们必须取、发射并且启动 4 条以上的指令。功耗和峰值成比例关系, 而性能则会保持持续速率 (为了降低功耗消费, 大多数最新的处理器采取了关闭处理器中不活跃部分的方法, 包括关掉芯片的部分时钟电源。尽管这类技术有一定的实用价值, 但是从长远来看它们无法阻止功耗效率的下降)。

此外,虽然推测是过去10年中最重要的开发指令级并行的技术,但是推测技术天生就是低效的。为什么?因为推测永远不可能是完美的;就是说,推测不可避免地会造成资源浪费,而在确定推测是否正确之前,我们无法判断推测是否加快了程序的执行速度。

尽管完美推测增加了一些额外的实现开销,但是由于它节省了静态功耗且加快了执行时间,因此完美的推测可以节省功耗。但是对于非完美的推测来说,由于错误推测和重置处理器状态需要额外的动态功耗,因此它会迅速地转变为能源低效的。鉴于实现推测所需的开销,包括寄存器重命名、重排序缓存、更多的寄存器等,对于绝大多数实际程序来说,推测处理器不可能节约能源。

那么如果我们集中精力提高时钟频率,情况又会怎样呢?遗憾的是,在提高时钟频率的过程中我们会遇到相似的问题:加快时钟频率会提高晶体管切换的频率,从而直接增大功耗。为了获得更高的时钟频率,需要有超长流水线的支持。而深度流水会带来额外的开销代价,同时使切换速率上升。

上述现象的最佳实例就是 Pentium III 和 Pentium 4 的对比。大致上来看, Pentium 4 可以说是 Pentium III 系统结构的深度流水版本。二者在功率消耗上的比大致与时钟频率的比相当。但遗憾的是, Pentium 4 对 Pentium III 的性能比要低于二者的时钟频率比,这是由于指令级并行的限制和代价造成的。

到这里,我们似乎已经找到了问题的答案——那就是开发指令级并行过程中的性能递减问题。过去几年中性能增长趋势减缓(见第1章),指令发射能力停滞不前,多核设计开始出现,在这种现象的背后我们可以看到这些限制的影子。我们将在结论部分继续讨论这个问题。

3.7 谬误和易犯的错误

谬误: 会有一种简单的方法使多发射处理器获得高性能,这种方法不需要在硅晶体面积上加大投入,也不会显著增加设计的复杂度。

前面几节的内容已经很明确地回答了这个问题。但令人吃惊的是,仍有相当数量的设计者相信这个谬误是正确的,他们投入了大量的精力试图找到这条捷径。尽管有可能设计一个相对简单的多发射处理器,但是当发射速率提高时,性能递减的现象会越来越明显,为实现宽发射而在硅晶体和能源上投入的代价会使在性能上得到的收益消耗殆尽。

除了硬件低效之外,为了开发更多的指令级并行度,处理器的编译技术也会变得相当复杂。编译器不仅要为一组复杂的转换提供支持,而且还需要对编译器进行校准,以使其能够在大多数基准测试程序上获得良好的性能,这个工作是相当困难的。

系统级的设计决策也会对性能产生影响,我们在上一节提到过,这种决策是非常复杂的。

陷阱: 只需改进多发射处理器的某一个方面,就可以获得整体性能的提高。

这个陷阱是对 Amdahl 定律的简单重述。设计者或许只会简单地看到设计的某一方面,比如较差的转移预测机制,然后对这个方面进行改进,并期望通过改进获得显著的性能提升。但是问题在于,限制多发射机器性能的因素有很多,只改进其中的一个方面往往会使其他一些新的限制因素暴露出来。

我们可以通过分析前面几节中给出的数据来理解这个问题。比如,研究图 3.3 中转移预测的影响,我们发现,与使用标准 2 bit 预测器相比,使用 tournament 预测器确实使 espresso 基准测试程序的并行度得到了显著提升(发射速度从 7 提高到 12)。但是如果处理器只提供 32 个重命名寄存器,则并行度将被限制在每时钟周期 5 条指令,即使更出色的转移预测机制也不会使情况得到改善。

3.8 结论

对于开发指令级并行来说,软件方法和硬件方法到底哪个更有价值?对这个问题的争论还在继续,但在过去几年中情况已有所改变。最初,软件方法和硬件方法是完全不同的,而且硬件方法所面临的复杂性问题一直为人所诟病。后来,一些拥有高时钟频率的高性能动态推测处理器的发展逐步打消了人们的疑虑。

IA-64系统结构以及Itanium设计的复杂性使设计者们得到启示,即软件方法已经不太可能使处理器更快(特别是对定点代码来说)、更小(在晶体管数量和核心面积方面)、更简单,以及在功耗方面更高效。有一点在过去五年里越来越明确,那就是在开发指令级并行和降低高性能处理器的复杂性和高功耗等方面,IA-64系统结构并没有取得质的突破。附录H将详细讨论这个问题。

宽发射在复杂性和性能递减上受到的限制可能会影响对同时多线程的应用。为验证最强大的同时多线程实现而制造一个宽发射处理器显然是不值得的。鉴于这个原因,现有的设计大多使用两个上下文的较为保守的同时多线程版本,或者使用两个上下文的简单多线程,这对简单的单发射和双发射处理器来说是一个恰当的选择。

系统结构的设计者们已经开始将注意力转移到在单芯片多处理器上实现线程级并行,而不是追求更多的指令级并行度。我们将在下一章中探讨单芯片多处理器的问题。2000年时,IBM推出了第一款商用单芯片通用多处理器,即IBM Power 4,它包含两个Power 3处理器和一个集成的二级Cache。此后,Sun Microsystems,AMD和Intel都开始将精力转移到单芯片多处理器上,而不再一味追求功能强大的单处理器。

到2005年时,指令级并行和线程级并行的平衡问题仍然存在,设计者们在各个方向上进行着探索,从单芯片多处理器上的简单流水线,到少数处理器上的指令级并行和同时多线程。在服务器市场上,能够开发更多线程级并行的选择也许是正确的;而在桌面市场上,单线程的性能仍然是首要需求。我们将在下一章继续讨论这个问题。

3.9 历史回顾和参考文献

随书光盘上的K.4节介绍了流水线和指令级并行的发展历程。我们为深入阅读和探讨这些主题提供了参考文献。

3.10 范例分析及习题^①

通过这个范例阐明以下概念:

- 软件相关引起的指令级并行的限制
- 在硬件资源约束下达到的指令级并行度
- 软件和硬件的相互作用引起的指令级并行的变化
- 权衡编译阶段的指令级技术与执行阶段的指令级并行技术

范例分析 1: 相关和指令级并行

本范例分析的目标是论证执行指令级并行过程中软件因素与硬件因素的相互作用。本范例分析给出了一个简洁的代码示例,并通过这个代码示例详细阐释了指令级并行的各种限制因素。对于在

^① 本范例分析及习题由Wen-mei W. Hwu和John W. Sias提供。

给定系统上执行的特定类型的代码，软件因素和硬件因素的相互作用是如何影响它的执行时间的呢？通过本范例分析，你可以对这个问题有一个直观的了解。

Hash表是一种组织大规模数据集合的流行数据结构，Hash表可以很容易地解决一些问题，比如查找数据集中值为100的元素等。在Hash表中，Hash函数根据数据元素的值生成一个函数值，数据元素根据这个函数值被分配给桶。桶中的数据元素以链表的方式进行组织，以某种顺序进行排序。在查找Hash表的过程中，首先需要确定要查找的数据值所对应的桶。之后遍历桶的链表中的数据元素，检查要查找的值是否在这些数据元素中。只要保证桶中的数据元素数足够少，就可以很快完成查找。

图3.14所示的C代码在Hash表中插入了N_ELEMENT个元素，Hash表的1024个桶以链表的形式组织，桶中的数据元素按值的升序排序。element[]数组包含要插入的元素。从第6行开始的外(for)循环的每个迭代插入一个元素。

```
1  typedef struct _Element {
2      int value;
3      struct _Element *next;
4  } Element;
5  Element element[N_ELEMENTS], *bucket[1024];
   /* 用要插入的数据项初始化数组元素;
      桶数组中的指针被初始化为NULL. */

6  for (i = 0; i < N_ELEMENTS; i++)
   {
7      Element *ptrCurr, **ptrUpdate;
8      int hash_index;

   /* 确定新元素要插入的桶. */
9      hash_index = element[i].value & 1023;
10     ptrUpdate = &bucket[hash_index];
11     ptrCurr = bucket[hash_index];
   /* 确定新元素的插入位置. */
12     while (ptrCurr &&
13            ptrCurr->value <= element[i].value)
14     {
15         ptrUpdate = &ptrCurr->next;
16         ptrCurr = ptrCurr->next;
17     }

   /* 更新指针以插入新元素 */
17     element[i].next = *ptrUpdate;
18     *ptrUpdate = &element[i];
   }
```

图 3.14 Hash 代码示例

图3.14中的第9行根据 $\text{element}[i]$ 的值计算 hash_index 的值, 即 Hash 函数值。这里使用的 Hash 函数非常简单; 它由元素数据值的低 10 位组成, 这个值通过元素数据值与 11 1111 1111 (十进制的 1023) 进行逻辑 AND 运算得到。

图 3.15 所示为我们在示例 C 代码中使用的 Hash 表数据结构。图 3.15 左侧的桶数组为 Hash 表。桶数组的每个入口包含一个指向链表的指针, 该链表保存桶的数据元素。如果桶 i 当前是空的, 则 $\text{bucket}[i]$ 包含的指针为 NULL。在图 3.15 中, 前 3 个桶各自包含一个元素, 其他的桶是空的。

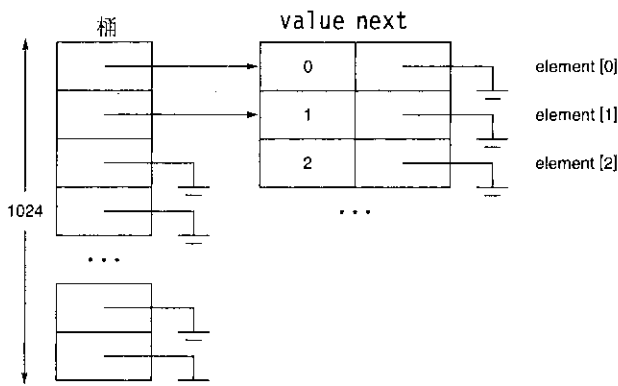


图 3.15 Hash 表结构

指针变量 ptrCurr 用来检查桶链表中的元素。在图 3.14 中的第 11 行, ptrCurr 被设定为 Hash 表中指定桶的第一个元素。如果由 hash_index 选中的桶是空的, 则对应的桶数组入口的指针为 NULL。

while 循环从第 12 行开始。第 12 行通过检查变量 ptrCurr 的内容来确定桶中是否还有要检查的元素。如果已经没有需要检查的元素, 则可能有两种情况: 或者桶是空的; 或者 while 循环之前的迭代已经检查了链表的所有元素, 这时第 13 行到第 16 行将被跳过。如果是第一种情况, 则新的数据元素将作为桶的第一个元素插入。如果是第二种情况, 则新的数据元素将作为链表的最后一个元素插入。

如果仍然有需要检查的元素, 则第 13 行将检查当前链表元素的值是否小于或等于要插入 Hash 表的值。如果是, 则 while 循环将继续检查链表的下一个元素; 第 15 行和第 16 行将 ptrCurr 指向链表的下一个元素。如果当前链表元素的值大于要插入 Hash 表的值, 则表明已经在链表中找到了新元素的插入位置; while 循环将中止, 新元素将被插入到当前 ptrCurr 指向的元素之前的位置。

变量 ptrUpdate 标识为了在桶中插入新数据元素而必须更新的指针。第 10 行将其设定为指向桶的入口。如果桶是空的, 则 while 循环将被跳过, 新数据元素将被插入, 这是通过将 $\text{bucket}[\text{hash_index}]$ 的指针从 NULL 改变为指向新数据元素而完成的, 如第 18 行所示。 while 循环结束后, ptrUpdate 指向为了在桶中的适当位置插入新数据元素而必须被更新的指针。

当执行退出 while 循环后, 第 17 行和第 18 行完成将新数据元素插入链表的最后工作。如果桶是空的, 则退出 while 循环时 ptrUpdate 指向 $\text{bucket}[\text{hash_index}]$, 这时 $\text{bucket}[\text{hash_index}]$ 为 NULL。第 17 行将 NULL 赋值给新数据元素的 next 指针。第 18 行将 $\text{bucket}[\text{hash_index}]$ 指向新数据元素。如果新数据元素小于链表中的所有元素, 则退出 while 循环时 ptrUpdate 也会指向 $\text{bucket}[\text{hash_index}]$, 这时 $\text{bucket}[\text{hash_index}]$ 为链表中的第一个元素。在这种情况下, 第 17 行将链表的第一个元素的指针赋值给新数据元素的 next 指针。

如果要插入的值大于链表中的部分数据元素, 而小于另一部分, 则 ptrUpdate 将指向插入位置前一个元素的 next 指针。在这种情况下, 第 17 行将使新数据元素的 next 指针指向插入位置的下一

个元素。第18行使插入位置之前的数据元素的 next 指针指向新数据元素。读者应当能够自己判断新数据元素插入链表末端的情况。

现在我们已经对这段C代码有了详细了解,接下来我们将分析这段代码中的可用指令级并行度。

3.1 [25/15/10/15/20/20/15]<2.1, 2.2, 3.2, 3.3, APP.H> 这部分将主要讨论在最佳执行情况下(即理想情况),运行时动态硬件调度器可用的指令级并行度(稍后,我们将讨论非理想情况下,动态硬件调度器和编译调度器的可用并行度)。对于理想情况,假设 Hash 表初始时是空的。设有 1024 个新数据元素,它们的值分别为 0 到 1023,因此它们将分别被分配给各自的桶(这简化了更新已知数组位置的问题)。图 3.15 所示为在这种理想状态下,插入前三个元素后 Hash 表的状态。由于理想状态下 `element[i]` 的值就是 `i`,因此每个元素都会被插入到自己的桶中。

在这个习题中,假设图 3.14 中的每一行代码需要花费一个执行周期,为了计算指令级并行,假设每一行代码需要一条指令。这些假设(不切实际的)只是为了在求解习题时简化代码记录而设定的。需要注意的是,为了测试循环是否应当继续,for 和 while 语句只在各自循环的迭代中执行。理想情况下代码序列中的很多相关都是不存在的,因此可以获得很高的指令级并行度。之后我们将讨论更一般的情况,这时一些实际的相关将限制可用的指令级并行度。

进一步假设代码是在理想处理器上执行的,即处理器拥有无限发射带宽、不受限制的重命名、完美的存储器二义性消除以及转移预测等,因此指令的执行只受到数据相关的限制。在上述环境下考虑下面的问题。

a. [25]<2.1>描述这段代码运行时硬件调度器可见的数据相关(真相关、反相关和输出相关)以及控制相关。标出实际的相关(即忽略访问不同地址的 load 和 store 之间的相关,即使编译器和处理器不能实际确定这一点)。在理想情况下,画出外循环连续 6 个迭代(插入 6 个元素)的动态相关图。在动态相关图中,需要标出动态指令实例之间的数据相关:根据循环的执行,源程序中的每条静态指令都有多个动态实例。提示:以下定义可以帮助你找到与指令有关的相关:

- 真数据相关: 哪些前序指令的结果要立即被指令引用?
- 反数据相关: 哪些后继指令对指令要读取的地址进行写操作?
- 数据输出相关: 哪些后继指令对指令要写的地址进行写操作?
- 控制相关: 哪些前序决定会影响指令的执行(在什么情况下可以达到)?

b. [15]<2.1>假设在上述理想情况下,使用你得到的动态相关图,有多少条指令被执行?用了多少个周期?

c. [10]<3.2>执行 for 循环过程中平均可用指令级并行度是多少?

d. [15]<2.2, App.H>在(c)中我们考虑了运行时硬件调度器可达到的最大指令级并行度。假设编译器知道它正在理想情形下工作,编译器怎样才能提高可用指令级并行度?提示:在理想情况下,哪些因素阻止了一次执行更多的迭代?怎样重组循环使其避开这些约束?

e. [25]<3.2, 3.3>为了简化起见,假设只有变量 `i`, `hash_index`, `ptrCurr` 和 `ptrUpdate` 占用寄存器。假设一般的重命名情况下,为了达到(b)中的最大并行度,需要多少寄存器?

- f. [25]<3.3>假设(a)的答案中每个迭代有7条指令。现在,假设对例子中的指令使用连续且状态稳定的调度,且发射率为每时钟周期3条指令,会对执行时间产生怎样的影响?
- g. [15]<3.3>最后,计算为了达到最大指令级并行度而需要的最小指令窗口大小。
- 3.2 [15/15/15/10/10/15/15/10/10/10/25]<2.1, 3.2, 3.3>现在让我们来考虑在非理想情况下,图3.14中的Hash表代码使用运行时动态硬件调度器进行调度时,可得到的指令级并行度(一般情况)。假设现在已经不能保证每个桶只会收到一个数据项。让我们在更实际的情况下,当增加了一些附加的、重要的相关后,重新估计和评价可用指令级并行度。
- 回想理想情况下,相对连续的内循环没有发挥作用,外循环已经提供了足够多的并行度。在非理想情况下,内循环将会发挥作用:while循环可能会执行一次以上。需要注意的是,内循环,即while循环,只有有限的并行度。这是由于:首先,while循环的每个迭代与前一个迭代的结果相关。其次,只有少量的指令被执行。
- 相比之下,外循环能够提供较多的并行度。只要外循环的两个元素被插入不同的桶,那么插入操作就可以并行进行。即使它们插入相同的桶,只要存储器二义性的消除可以保证每个元素的load和store的正确性,那么它们仍然可以并行执行。
- 需要注意的是,数据元素的值可能会随机分布。尽管我们希望为读者提供更接近实际的执行场景,但我们需要从更有规律而不实际的数据值模式开始,以方便我们进行分析。这种方式通过提供中间步骤,来理解最一般的随机数据值模式。
- a. [15]<2.1>当要插入的数据元素的值为0, 1, 1024, 1025, 2048, 2049, 3072, 3073, ...时,画出图3.14所示代码的动态相关图。当while循环执行一次和一次以上时,描述for迭代之间的新的相关。特别要注意的是,while循环现在可能会执行一次以上。而外部的for循环的指令数可能会因此变化。为了确定load和store之间的相关,假设消除动态存储器二义性不能解决基于不同基址寄存器的存储器访问之间的相关。例如,运行时硬件不能消除基于ptrUpdate的store和基于ptrCurr的load之间的相关。
- b. [15]<2.1>根据你在(a)中画出的动态相关图,有多少条指令被执行?
- c. [15]<2.1>根据你在(a)中画出的动态相关图,假设硬件资源无限的情况下,要执行(b)中计算结果的指令数,需要花费多少个时钟周期?
- d. [10]<2.1>根据你在(a)中画出的动态相关图,有多少可用指令级并行度?
- e. [10]<2.1, 3.2>假设运行时存储器二义性消除机制同(a)中相同,标出引起最坏情形的数据元素序列,在这种情形下相关会影响可用指令级并行度。
- f. [10]<2.1, 3.2>现在,假设在(e)所述的最坏情形序列中,阐述完美运行时存储器二义性消除机制的影响(即系统跟踪所有的外部store,允许所有不冲突的load继续执行)。在动态相关图中,导出执行所有指令所需的时钟周期数。
- 以你目前掌握的知识为基础,考虑下面的定性分析问题:如果在已知前序store指令地址之前允许推测发射load指令,这会带来怎样的效果?在这段代码中,这类推测会对存储器时延的重要性产生怎样的影响?
- g. [15]<2.1, 3.2>继续(f)中的假设,计算执行的指令数。
- h. [10]<2.1, 3.2>继续(f)中的假设,计算运行时硬件可用的指令级并行度。
- i. [10]<2.1, 3.2>在(h)中,有限的指令窗口大小会对指令级并行度产生怎样的影响?
- j. [10]<3.2, 3.3>现在,继续考虑(h)中的解答,描述转移预测缺失以及每个转移预测对可用并行度的影响。简要介绍功耗和效率的含义。执行多路推测的潜在代价和好处是什么?

(即初始化执行一些指令,但由于错误的转移预测,这些指令稍后将被撤销)?提示:考虑在插入点之前,错误预测元素数目的情况下执行后继插入操作所产生的影响。

k. [25]<3>最坏情况下的相关会限制编译器的调度和优化,静态相关图捕捉全部这些相关。

画出图 3.14 所示 Hash 表代码的静态相关图。

将静态相关图与前面讨论的各种动态相关图进行比较。比较用静态方法和动态方法在此例的 Hash 表代码中开发指令级并行度的情况,用一两段文字解释意义。特别是,如果不得不持续地考虑最坏情况,会给编译器带来怎样的约束,而在相同情况下,硬件机制则有可能利用偶然情况的机会避免这些约束。哪种排序方式会使编译器更好地利用这段代码?

第4章 多处理器和线程级并行

传统系统结构的变革发生在20世纪60年代中期,此时,提高计算机运行速度所能得到的改进已经变得越来越少……电子电路的速度最终将受到光速的限制……而大量的电路已经运行在纳秒级别了。

——W. Jack Bouknight 等
The Illiac IV System [1972]

我们未来产品发展的主要目标就是多核设计。我们相信它对于计算机工业来说是一个关键的转折点。

——Intel 公司总裁 Paul Otellini
在2005年Intel开发者讨论会上作的关于Intel未来发展方向的报告

4.1 简介

正如本章开始时的引语所述,许多年前开始就有部分研究人员认为单处理器系统结构的发展很快就要接近尾声了。但显然他们的观点是不成熟的,因为我们观察到从1986年到2002年这段时间里,在微处理器的推动下,单处理器性能的增长达到了自从20世纪50年代后期和60年代初期第一个晶体管计算机诞生以来的最高速度。

尽管单处理器仍在发展,但多处理器在20世纪90年代显得越来越重要。设计者们在寻找拥有高性价比优势的商业计算机的过程中,发明了一种服务器和超级计算机的制造方法,它比单一的微处理器具有更高性能。就像我们在第1章和第3章中提到的,由于开发ILP的空间正在减少,再加上对电源关注程度的增加,单处理器发展的速度正在逐步减慢,这最终导致了计算机系统结构的一个新时代的到来,即多处理器唱主角的时代。这也就是本章开头第二条引语所要传达的信息。

除此之外,由于以下几点因素,对于多处理系统的依赖趋势显得愈发明显:

- 对服务器及其性能关注的不断增加。
- 以数据为中心的应用不断增多。
- 台式计算机性能(至少除了图形性能以外)的增长似乎不再重要。
- 对如何有效利用多处理器的理解的不断深入,尤其是在严重依赖线程级并行的服务器环境下。
- 工业制造中批量复制的方法比专门生产的方法更具成本优势,而所有的多处理器设计正好享有这种优势。

尽管如此,仍然有两个问题需要解决。第一,多处理器系统结构是一个大且复杂的领域,其中很多领域仍处于不成熟的阶段。新的想法层出不穷,而直到最近,也还是以失败的系统结构居多。要对多处理器的设计以及权衡策略进行全面的论述恐怕需要另一本专著才能完成(而Culler, Singh和Gupta[1999]在他们长达1000页的著作中仅仅只讨论了多处理器而已);其次,全面的论述必然要对一些经不起时间考验的方法进行讨论,这正是本书所要避免的。

由于这些原因,我们宁愿将注意力集中在多处理器设计的主流技术上,即由少量到中等数量的处理器(4~32个)组成的多处理器设计。这样的设计由于处理器数量和价格的原因而占据主导地位。本书中将用不多的篇幅(附录H)对大规模多处理器(≥ 32 个)的设计进行论述,附录H涵盖了设计这种处理器的诸多方面,以及并行科学计算的性能表现,这是大规模多处理器应用的主要类别。在大规模多处理器中,互连网络的设计是至关重要的;附录E中将主要讨论这一主题。

并行系统结构的分类

我们首先给出一个分类方法,这样读者就能够据此识别多处理器系统的各种不同设计方案以及多处理器主流技术的演变背景。我们会简要介绍这些设计方案及其基本原理;有关这些不同模型产生的原因,将在附录K中论述。

使用多个处理器来提高性能和增加可用性的思想应追溯到最早的电子计算机。大约30年前,Flynn[1966]提出了一种对所有计算机进行分类的简单模型,在今天看来这个模型仍然很有价值。根据多处理器中限制要求最多的单元中的指令所调用的数据流和指令流的并行度,他把所有的计算机归为四类:

1. 单指令流,单数据流(SISD):单处理器。
2. 单指令流,多数据流(SIMD):同一条指令被多个使用不同数据流的多处理器执行。SIMD计算机通过将相同的操作以并行的方式应用于数据的各个项来实现数据级的并行。每个处理器有自己的数据存储器(因此是多种数据),但系统中有唯一的指令存储器和控制处理器,用来获取和分配指令。对于有明显的数据级并行机制的应用来说,SIMD方法是十分高效的。我们在附录B和C中提到的多媒体扩展就是SIMD并行的一种形式;附录F提到的向量处理器系统结构是这种系统结构中最大的一个分支。在过去的几年里,随着图形性能重要性的提高,特别是游戏市场的扩大,SIMD方法再度被广泛应用。要达到构建三维、实时的虚拟环境所需要的理想性能,最好使用SIMD方法。
3. 多指令流,单数据流(MISD):至今还没有这种类型的商用机器。
4. 多指令流,多数据流(MIMD):每个处理器取自己的指令并对自己的数据进行操作。MIMD计算机实现线程级并行,因为多个线程是按并行操作的。一般来说,这种线程级并行比数据级并行更加灵活,因此用途也更为广泛。

这只是一个粗略的分类,有些机器是上述不同类型的混合体。但是,这种分类方法可以为设计制定一个框架。

因为MIMD模型可以实现线程级并行机制,所以就成为一般多处理器设计所选择的系统结构,这同时也是本章的关注点。导致MIMD多处理器格外引人注目的原因还有两个:

1. MIMD灵活性强。在必要的软件和硬件支持下,MIMD既能作为单用户多处理器为单一应用程序提供高性能(向量处理器除外),又可作为同时运行多个任务的多道程序多处理器系统使用,甚至还可以提供结合这两种任务的应用。
2. MIMD能够充分利用现有微处理器的性价比优势。实际上,当今几乎所有商用多处理器系统所使用的微处理器与工作站及单处理器服务器所使用的微处理器都是相同的。此外,多核芯片通过复制方式可以有效降低单处理器内核的设计成本。

集群是一种流行的MIMD计算机类型,它通常使用标准组件和标准网络技术,这样就可以尽可能多地支持多种通用技术。在附录H中我们区分了两种不同的集群:商业集群,它主要依赖于第三方处理器和互连技术;客户集群,客户可以定制详细节点设计或互连网络,或者对其同时定制。

在商业集群上，集群的节点通常是刀片服务器或机架服务器（包括小规模的多处理器服务器）。对于强调吞吐量和几乎不要求线程间通信的应用，像Web服务、多程序以及一些事务处理的应用，可以被移植到集群上，而且代价也不高。商业集群常常由用户或计算机中心装配，而不由制造商装配。

客户集群则主要针对并行应用，即对一个简单的问题可以使用大量的并行机制。这种应用要求计算之间的大量交互，并需要在集群上定制节点和互连设计以达到更高的效率。目前，最大和最快的多处理器系统就是客户集群，比如IBM Blue Gene（将在附录H中讨论）。

从20世纪90年代开始，单芯片日益增长的容量允许设计者将多个处理器设计在一块单独的晶片上。这种方法一开始称为片内多处理器或单芯片多处理器，现在称为多核，这个名字来自于在单独一个晶片上设计多个处理器内核的方法。在这样的设计中，多个内核共享一些资源，比如二级或三级Cache、存储器和I/O总线。比较新的一些处理器，包括IBM Power 5，Sun T1以及Intel Pentium D和Xeon-MP，都是基于多核和多线程的。就像在一个多处理器上可以通过复制来降低设计成本一样，多核可以更多地依赖复制而不是构造超标量系统结构来降低成本。

对于MIMD，每个处理器执行自己的指令流。在许多情况下，每个处理器运行不同的进程。进程是可以独立运行的一段代码，进程的状态包含了处理器运行这个程序的所有必要信息。在一个多道程序环境中，各个处理器可能执行相互独立的任务，因此，每个进程与在其他处理器上执行的进程不相关。

让多个处理器执行同一个程序并且共享程序的代码和地址空间也是十分有效的。用这种方式共享代码和地址空间的多个进程，通常称之为线程。现在，线程经常用来指运行在不同处理器上的多个执行过程，即便它们并没有共享同一地址空间。例如，多线程的系统结构实际上允许不同地址空间上的多个进程同时执行，而且也允许共享地址空间上的多个线程同时运行。

要充分利用包括 n 个处理器的MIMD多处理器，通常必须执行至少 n 个进程或线程。一个进程中的各个独立线程通常由程序员标识或由编译器创建。这些线程可能来自于大规模且各自独立的进程，由操作系统调度和管理，但一个线程也可能由一个循环的多次重复执行过程构成。虽然分配给线程的计算量（称为粒度大小）在高效开发线程级并行机制时十分重要，但它与指令级并行机制的本质区别是，线程级并行机制在高层次上被软件系统所标识，且线程由几百条至几百万条并行执行的指令组成。

虽然比SIMD计算机的开销要高，但线程同样可以用来开发数据级并行。开销意味着粒度必须足够大以实现高效的并行机制。例如，虽然一个向量处理器（见附录F）可以高效地并行执行短向量操作，但当并行机制被划分给各个线程时所导致的粒度很小，以至于开发并行机制的开销有限。

现有的MIMD机器根据处理器个数可分为两类，这两类机器的存储器组织方式和互连策略也不同。由于处理器的构成方式会随时间而变化，因此，根据存储器组织方式而不是处理器构成方式来区分多处理器的种类更为合适。

第一种类型称为集中式共享存储器系统结构。在2006年这种系统结构的机器最多拥有几十个处理器（少于100个内核）。对于处理器数目较少的多处理器，各个处理器可以共享单个集中式存储器。在使用大容量Cache的情况下，单一存储器（可能是多组）能够确保小数目处理器的存储访问得到及时响应。通过使用多个点对点的连接，或者通过交换机，再加上额外的存储器组，集中共享存储器设计可以扩展到几十个处理器。即使设计更大规模的集中共享存储器的多处理器在技术上行，但是随着处理器数目的增多，这种共享单个集中式存储器的方案前景不被看好。

由于只有单一存储器，它对每个处理器而言都是对等的，并且每个处理器的访问时间相同，这种多处理器系统也称为对称（共享存储器）多处理器系统（SMPs），这种系统结构也称为均匀存储器访问（UMA），这是因为所有的处理器访问存储器都有相同的时延（latency），即使存储器是按多

方式组织的。
本书在4.2节将

图4

第二
构。为了
存储器在
处理器文
少。当然
接互连

组方式组织的。集中式共享存储器系统结构的组成如图4.1所示。它是迄今为止最流行的系统结构。本书在4.2节将详细讨论这种多处理器的系统结构。

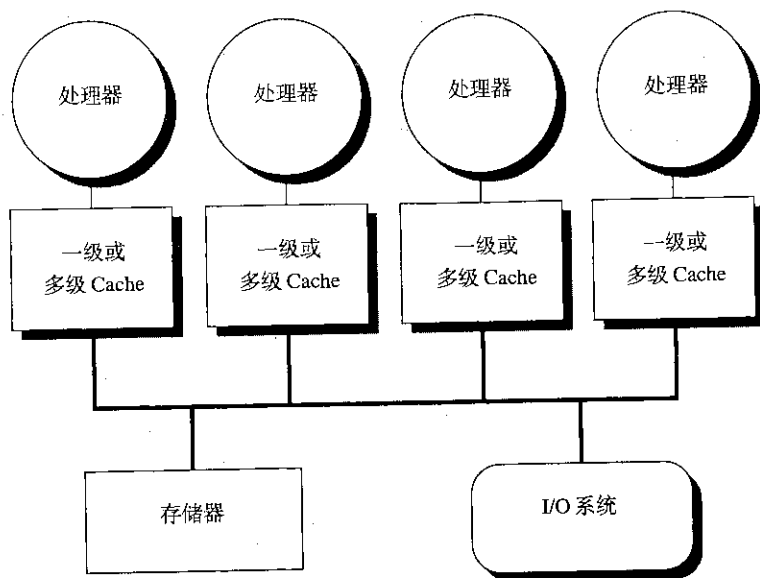


图4.1 集中式共享存储器的基本结构。多个处理器-Cache子系统共享同一个物理存储器，典型方式是通过一个或更多的总线或一个交换机连接。此系统结构的一个关键特性是所有的处理器访问存储器的时间一致

第二种类型的多处理器的存储器在物理上是分布的。图4.2给出了这种类型多处理器的组成结构。为了支持更多的处理器，存储器不能按照集中共享方式组织，而必须分布于各个处理器；否则，存储器在为多个处理器提供所需要的带宽时将无法避免较长的时延。随着处理器性能的迅速提高及处理器对存储器带宽需求的增加，使用分布式存储器系统结构的多处理器系统的处理器数目正在减少。当然，大量的处理器要求互连网络必须有足够的带宽，我们将在附录E中给出有关的例子。直接互连网络（如交换机）和间接互连网络（如多维网状网）都有可能用到。

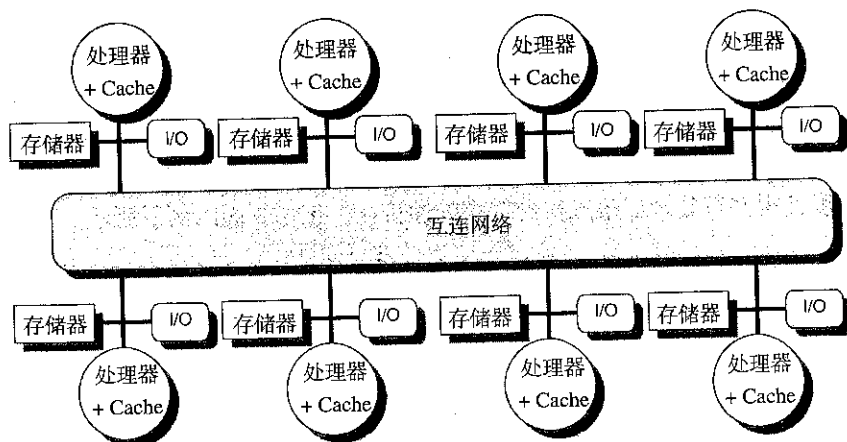


图4.2 分布式存储器多处理器的基本结构由多个独立节点构成。其中每个节点包含处理器、存储器、输入输出系统和互连网络的接口，各个节点通过互连网络连接在一起。每个节点可能含有少量处理器，这些处理器使用小总线或其他互连技术连接在一起。节点内采用的互连技术的可扩展性低于全局互连网络

将存储器分布到各个节点上有两个主要的好处。第一，如果大部分访问是在节点内的本地存储器中进行的，这样做是增大存储器带宽比较经济的方法。第二，缩短了本地存储器访问的时延。在处理器变得越来越快并要求存储器带宽更高以及存储器时延更低的情况下，这两个优点使得这种方法在构建较少处理器的系统时颇具吸引力。分布式存储器系统结构的主要缺点是由于处理器不再共享单一集中存储器，处理器间的数据通信在某种程度上变得更加复杂，且时延也更大。下面会看到使用分布式存储器系统结构时，有两种进行处理器间通信的不同方式。

通信和存储器系统结构模型

如前所述，大规模多处理器系统结构使用与各个处理器分布在一起的多个存储器。根据处理器间传递数据所用的方法，有两种不同的系统结构。

第一种方法通过共享的地址空间进行通信。物理上分开的存储器能够作为逻辑上共享的地址空间进行寻址，就是说只要有正确的访问权限，任何一个处理器都能够通过引用地址的方式访问任意节点上的存储器。这类机器称为**分布式共享存储器（DSM）**系统。所谓**共享存储器**指的是共享寻址空间，就是说，两个处理器中相同的物理地址指向存储器中的同一个位置。共享存储器并不是说有一个单一的、集中式的存储器。与对称式共享存储器多处理器——也称UMA（均匀存储器访问）相比，DSM多处理器由于访问时间取决于数据字在存储器中的位置，因而也称为NUMA（非均匀存储器访问）。

另外一种地址空间由多个私有的地址空间组成，这些私有地址空间在逻辑上是分散的，并且不能被远程处理器寻址。在这种机器中，两个不同处理器中相同的物理地址分别指向两个不同存储器中的不同位置。每个处理器-存储器模块本质上是一台独立的计算机。最初，这种计算机由不同的处理节点和专用的互连网络组成。目前，这种类型的大多数设计实际上就是**集群**（附录H中讨论）。

每一种地址空间组织方式都有相应的通信机制。对于共享地址空间的机器，可以利用地址空间通过load和store操作隐式地传递数据；所谓的**共享存储器**就是由此得名的。对于有多个寻址空间的多处理器系统，数据通信通过显式地在处理器间传送消息来完成。因此，这类机器经常称为**消息传递多处理器系统**，集群就是使用消息传递的一类系统。

并行处理遇到的挑战

多处理器的应用非常广泛，不论是运行相互之间没有通信的独立任务，还是必须通过线程通信才能完成的并行程序，都可以应用多处理器。然而，有两个障碍使得并行处理的应用遇到了挑战。这两个障碍都可以用Amdahl定律解释。障碍的难易程度是由应用和系统结构共同决定的。

第一个障碍是程序可获得的并行度是有限的。第二个障碍来自于通信相对较高的开销。只能得有限的并行度使得并行处理器很难得到较高的性价比，正如下面的例子所示。

例题 假设要用100个处理器获得80倍的加速比。那么原来的计算中串行部分该占多大比例呢？

解答：Amdahl定律是

$$\text{加速比} = \frac{1}{\text{改进部分所占比例} + (1 - \text{改进部分所占比例}) \times \text{改进部分的加速比}}$$

为简单起见，在本例中假设程序仅有两种执行模式：一种是使用所有处理器的并行模式，就是改进模式，另一种是仅利用一个处理器的串行模式。在这种简化下，改进部分的

就简化为处理器个数，而改进模式所占的比例就是在并行模式中花费的时间。代入上面的公式

$$80 = \frac{1}{\frac{\text{并行部分所占比例}}{100} + (1 - \text{并行部分所占比例})}$$

对等式简化得到

$$0.8 \times \text{并行部分所占比例} + 80 \times (1 - \text{并行部分所占比例}) = 1$$

$$80 - 79.2 \times \text{并行部分所占比例} = 1$$

$$\text{并行部分所占比例} = \frac{80 - 1}{79.2}$$

$$\text{并行部分所占比例} = 0.9975$$

因此要用 100 个处理器得到 80 的加速比，原程序中只能有 0.25% 的部分是串行的。当然，要得到线性加速比（即用 n 个处理器得到加速比为 n ），整个程序必须没有串行部分，全部是并行的。实际上，程序并不只是在完全并行或串行的模式下运行，通常并行模式中并没有使用全部的处理器，而仅仅使用了一部分处理器。

第二个主要挑战与并行处理器中远程访问的时延较长有关。现有共享存储多处理器中，处理器间的数据通信少则花费 50 个时钟周期（多核），多则超过 1000 个时钟周期（大规模多处理器），时间的长短由通信机制、互连网络的类型和多处理器的规模决定。长通信时延的影响非常之大，让我们来看一个简单的例子。

例题 假设有一个应用程序在一台 32 个处理器的多处理器系统上运行，该处理器访问一个远程存储器需要 200 ns。对于这个应用，假设除了涉及通信的存储器访问外，所有访问都命中本地存储系统（这样假设有点乐观）。执行远程访问时处理器会阻塞，处理器的时钟频率为 2 GHz。如果基本 CPI（假设所有的访问命中 Cache）是 0.5，那么多处理器在没有远程访问时比只有 0.2% 的指令涉及远程访问时能快多少？

解答：首先计算 CPI，有 0.2% 远程访问的多处理器的 CPI 是

$$\begin{aligned} \text{CPI} &= \text{基本 CPI} + \text{远程请求率} \times \text{远程请求开销} \\ &= 0.5 + 0.2\% \times \text{远程请求开销} \end{aligned}$$

远程请求开销是

$$\frac{\text{远程请求开销}}{\text{周期时间}} = \frac{200 \text{ ns}}{0.5 \text{ ns}} = 400 \text{ 周期}$$

然后我们可以计算 CPI：

$$\text{CPI} = 0.5 + 0.8 = 1.3$$

全部为本地调用的多处理器将会快 $1.5/0.3 = 2.6$ 倍。实际的性能分析会更加复杂，因为有些非远程访问可能会在本地存储器系统层次中缺失，并且远程访问的时间也不一定会是固定值。例如，远程访问的开销可能会更大，因为使用全局互连网络的多个远程访问引起的竞争会导致时间不断增加。

这些问题——并行度低和远程通信时延太长——是使用多处理器的两个最大挑战。只有在软件中采用更好的并行算法才能克服并行度低的问题。要减少长时间远程访问的时延,可以通过系统结构实现,也可以通过程序员实现。例如,在硬件上缓存共享数据,或者在软件上重新构造数据就能增加本地访问,因而也就减少了远程访问的频率。还可以使用多线程(在第3章和本章后面讨论)或预取(在第5章和本章后面讨论)来减少时延的影响。

本章大部分内容集中讨论减少长通信时延的技术。例如,4.2节和4.3节将讨论在确存储器一致性的前提下,如何使用Cache来减少远程访问频率。4.5节讨论同步,因为它本质上涉及到处理器间的通信,因此它也是潜在的性能瓶颈。4.6节讨论时延隐藏技术和共享存储的存储器一致性模型。在附录I中我们主要关注大规模多处理器,它在科学计算中占据统治地位。我们将阐述这种应用的本质,以及在使用几十到上百个处理器的情况下,要获得一定的加速比将面临的挑战。

要理解现代的共享存储器多处理器,首先需要掌握有关Cache的基本知识。读者可以参考我们的介绍性教材*Computer Organization and Design: The Hardware/Software Interface*。如果像写回Cache和多级Cache这样的概念也不清楚,读者应该首先花一些时间复习附录C。

4.2 对称式共享存储器系统结构

大容量、多层次的Cache能够大量减少单个处理器对存储器带宽的要求。如果单个处理器对存储器的带宽要求减少了,多个处理器就能共享同一个存储器。从20世纪80年代开始,随着微处理器逐步成为市场主流,这一观点促使许多设计者制造出小规模多处理器系统。在这些系统中,几个微处理器通过总线共享同一个物理存储器。因为这些处理器体积较小,并且大容量Cache明显减少了总线带宽的需求,所以在存储器带宽充足的情况下,这种机器极为划算。在这种机器的早期设计阶段,设计者们把整个处理器和Cache子系统放置在一块电路板上,然后再将这块电路板插在总线上。到20世纪90年代,其后续的版本已经能够在每块电路板上放置两个或四个处理器了,而且常常会使用多种总线和交叉存取的存储器以支持更快的处理器。

IBM公司在2000年首先将第一个片内多处理器推向通用计算机市场。AMD和Intel公司紧随其后于2005年在服务器市场各自推出了双处理器版本的产品, Sun公司在2006年推出8处理器多核的T1。4.8节将介绍T1的设计和性能,前面的4.1节已经给出了这种多处理器的一个简单图表(见图4.1)。目前,最新的高性能处理器对存储器需求已经超过了合理的总线能力,其结果是导致最新的设计已经开始使用小规模交换机或受限的点对点网络。

对称式共享存储器系统支持共享和私有数据的缓存。私有数据被单个处理器使用,而共享数据则被多个处理器所使用,基本上是通过读写共享数据完成处理器之间的通信。把一个私有数据缓存之后,对该数据的访问就可以在Cache中进行,这样就减少了平均访问时间和对存储器带宽的需求。因为没有其他处理器使用这些数据,程序的行为与单存储器系统相同。当共享数据装载到Cache中时,会在多个Cache中形成副本。这样做除了会减少访问时延和降低对存储器带宽的要求外,还能减少多个处理器同时读取共享数据时的竞争现象。然而,把共享数据放入Cache又出现了一个新的问题:Cache一致性。

什么是多处理器的Cache一致性

遗憾的是,缓存共享数据带来了一个新的问题,这是因为两个不同的处理器所保存的存储器视图是通过各自的Cache得到的,如果没有其他的防范措施,则会导致两个处理器分别得到两个不同

的值。图4.3说明了这个问题,并解释了为什么两个不同的处理器对存储器相同位置进行操作会得到不同的数值。这个问题常常称为 **Cache 一致性问题**。

时间	事件	处理器 A 的 Cache 内容	处理器 B 的 Cache 内容	存储器位置 X 的内容
0				1
1	A 读 X	1		1
2	B 读 X	1	1	1
3	A 向 X 写入 0	0	1	0

图 4.3 Cache 一致性问题: 两个处理器 (A 和 B) 对同一个存储器位置 X 进行读写操作。假设最初两个 Cache 都不包含该变量而且 X 的值为 1。假设是写直达 Cache; 而且写回 Cache 会带来更加复杂的情况。当 X 的值被 A 改写后, A 的 Cache 和存储器中的副本都做了更新, 但 B 的 Cache 则没有, 如果 B 读取 X, 得到的值为 1

一般情况下, 如果在一个存储器系统中读取任何一个数据项的返回结果总是最近写入的数值, 那么就可以认为该存储器具有一致性。这个定义尽管看起来是正确的, 但仍很模糊且简单; 实际情况要复杂得多。这个简单的定义包括了存储器系统行为的两个不同方面, 它们对于编写正确的共享存储程序都是至关重要的。第一个方面称为**一致性 (coherence)**, 它定义了读操作可以返回什么样的数值。第二个方面称为**连贯性 (consistency)**, 它定义了写入的数值什么时候才能被读操作返回。我们先看一下一致性问题。

如果一个存储器系统满足如下条件, 那么认为该存储器系统是一致的:

1. 处理器 P 对地址 X 的写操作后面紧跟着处理器 P 对 X 的读操作, 而且在这次读操作和写操作之间没有其他处理器对 X 进行写操作, 这时读操作总是返回 P 写入的数值。
2. 在其他处理器对 X 的写操作后, 处理器 P 对 X 执行读操作, 这两个操作之间有足够的间隔而且没有其他处理器对 X 进行写操作, 这时, 读操作返回的是写入的数值。
3. 对同一个地址的写操作是**串行执行的**; 也就是说, 任何两个处理器对同一个地址的两个写操作在所有处理器看来都有相同的顺序。例如, 如果向同一个地址中先后写入数值 1 和数值 2, 处理器决不会从该地址中先读出 2 再读出 1。

第一个性质是为了保证程序的顺序——即使在单处理器中也要保证这个性质。第二个性质定义了什么意味着有一个一致性的视图: 如果一个处理器对某个数据项执行读操作时, 总是读入旧的数值, 我们就认为这个存储器是非一致的。

写操作串行化的要求更加细致, 且同等重要。如果没有把写操作串行化, 而处理器 P1 写入地址 X, 紧接着 P2 写入地址 X, 那后果会怎样? 将写操作串行化保证了每个处理器都能在某个时间看到 P2 写入的结果。如果没有把写入操作串行化, 就会出现一些处理器先看到 P2 的写入结果再看到 P1 的写入结果, 从而可能保留 P1 写入的数值。避免这种情况出现的最简单方法是将写操作串行化, 这样对同一个地址的写操作在所有处理器看来就具有相同的顺序了; 这个性质称为**写串行化**。

虽然上述的三个性质足以保证一致性, 但写入的数据什么时候才可以读取也是至关重要的。因为我们不能要求由其他处理器写入 X 的数值后, 对 X 的读操作能立即看到该值。例如, 如果某个处理器对 X 的写操作只领先其他处理器对 X 的读操作很小一段时间, 那么无法保证该读操作能返回写入的数值, 因为这一刻写入的数据甚至可能还没有被处理器发送出去。**存储器连贯性模型**定义了写入的数值必须在什么时候才能被读操作读出的问题——4.6 节会讨论这个问题。

一致性和连贯性是互补的：一致性定义了对同一个存储器地址进行的读写操作行为，而连贯性定义了关于访问其他存储器地址的读写操作。我们做两个假设：一是直到所有处理器都看到了写操作的结果之后一个写操作才算完成，并且后续的写操作才可以开始；二是处理器不会因为其他存储器操作而改变写操作的顺序。这两个条件意味着如果处理器向地址A写入后又向地址B写入，所看到B中新值的处理器必须也能看到A的新值。这些限制允许处理器重新安排读操作的顺序，但要求处理器按照程序规定的顺序执行写操作。4.6节之前各节中的内容都是基于这个假设的，在4.6节中我们将会给出这个定义的确切含义，以及其他一些可供替代的选择。

实施一致性的基本方案

多处理器和输入输出的一致性问题的起源是类似的，但各自还是具有一些不同的特征，这些特征会影响相应的解决方案。输入输出中，很少出现一个数据有多个副本的情况——这是要尽量避免的，而多处理器系统中的情况恰恰相反，在多个处理器系统上运行的程序会要求在多个Cache中有同一个数据的副本。在支持Cache一致性的多处理器系统中，Cache提供共享数据的迁移和复制。

数据项可以迁入本地Cache并以透明的方式使用，这是一致性的Cache所提供的数据迁移功能。这样不但能减少访问远程共享数据项的时延，而且可以减少对共享存储器带宽的需求。

因为Cache在本地为那些被同时读取的共享数据做了备份，所以一致性的Cache也要为这些数据提供副本。而副本可以减少访问时延和读取共享数据时的竞争现象。对这种迁移和复制的支持对于访问共享数据的性能来说是至关重要的。为了维护Cache一致性，小规模多处理器系统并不是通过软件而是通过在硬件上引入一个协议解决这一问题的。

这个用于维护多个处理器一致性的协议称为Cache一致性协议（cache-coherence protocols）。实现Cache一致性协议的关键在于跟踪所有共享数据块的状态。目前广泛采用的有两类协议，它们采用不同的技术跟踪共享数据：

- **目录式**：把物理存储器的共享状态存放在一个地点，称之为目录。我们会在4.4节中讨论可扩展共享存储器系统结构时集中讨论这个方法。目录式一致性比监听式的实现开销略微偏高，但是可以用来扩展更多的处理器。Sun公司的T1设计就采用了目录式，尽管它拥有集中式物理存储器。
- **监听式**：每个含有物理存储器中数据块副本的Cache还要保留该数据块共享状态的副本，但是并不集中地保存状态。Cache通常可以通过广播媒介（总线或交换机）访问，所有的Cache控制器对总线进行监视或监听，来确定它们是否含有总线或交换机上请求的数据块的副本。本节集中讨论这种方法。

由于多处理器系统中使用微处理器，及连接到单一共享存储器的Cache，这使得监听协议用得越来越多。因为这种协议能使用已经存在的物理连接——通往存储器的总线——来查询Cache的状态。在下面一节中我们将会介绍基于监听的Cache一致性在共享总线上的实现方法，但是任何向所有处理器广播Cache缺失的通信媒介都可以用来实现基于监听的一致性。这种向所有Cache提供广播的方法使得其实现变得更简单，但也限制了其可扩展性。

监听协议

有两种方法可以保证上面所说的一致性。一种是在处理器写数据项之前保证该处理器能独占地访问数据项。这种协议称为**写无效协议**，因为它在执行写操作时要使其他副本无效。而且这种协议是到目前为止在监听和目录方案中最常用的协议。独占访问确保写操作执行后不存在其他可读或可写的数据项副本：Cache 中该数据项的所有其他副本都是无效的。

处理器活动	总线活动	处理器 A 的 Cache 内容	处理器 A 的 Cache 内容	存储器 X 位置 Cache 的内容
				0
处理器 A 读 X	Cache 缺失于 X	0		0
处理器 B 读 X	Cache 缺失于 X	0	0	0
处理器 A 向 X 写 1	对 X 无效	1		0
处理器 B 读 X	Cache 缺失于 X	1	1	1

图 4.4 监听总线方式的写无效协议实例，使用写回式 Cache。假设两个 Cache 最初都没有 X，并且存储器中 X 的值为 0。处理器和存储器中的内容是在处理器和总线活动全部完成后的数值。空格表明没有动作或没有存放副本。当 B 发生第二次缺失时，处理器 A 会应答，同时取消来自存储器的响应。然后，B 的 Cache 和存储器 X 的内容都得到更新。这种当块共享时对存储器进行更新的做法简化了协议实现，但也可能只有当块被替换时才能跟踪所有权并强制写回。这就要求引入一种称为“所有者”的额外状态，它表示块可以被共享，但是在块被改变或替换时，由所有者处理器负责更新其他处理器和存储器

图 4.4 给出了一个基于监听总线的写无效协议的例子，其中使用了写回式 Cache。为了说明一致性的具体实现过程，下面分析写操作后面紧跟其他处理器执行读操作的情况：由于写操作要求独占访问，执行读操作的处理器所保留的任何副本都被置为无效（协议正是由此而得名）。因此，执行读操作时，可能会发生 Cache 缺失，然后要取回新的数据副本的情况。对于写操作，我们要求执行写操作的处理器独占访问，这样就防止了任何其他处理器同时执行写操作的情况。如果两个处理器试图同时对同一个数据项执行写操作，它们中只有一个会在竞争中获胜（下面会给出如何确定谁能获胜），这样另外一个处理器的副本就被置为无效。竞争失败的处理器要完成写操作，就必须首先取得新的数据副本，而这个副本中已经包含了更新后的数据。所以，这个协议强制执行了写操作的串行化。

除了写无效协议，还有另一种方法：写入数据项时更新该数据项所有的副本。这种类型的协议称为**写更新或写广播协议**。因为写更新协议必须将所有的写操作广播给共享 Cache，所以需要更大的带宽。由于这个原因，所有近期的多处理器都选择执行写无效协议，我们会在本书剩下的部分重点关注这个协议。

基本实现技术

在小规模多处理器系统中实现写无效协议的关键是使用总线或其他广播媒介来完成无效操作。要实现无效操作，处理器只要取得总线控制权然后在总线上广播无效数据的地址即可。所有的处理器都要不断地监听总线来监测地址。处理器要检查各自的 Cache 中是否有总线上广播的地址。如果有，则 Cache 中相应的数据要置为无效。

当向一个共享块执行写操作时，写处理器必须获得对总线的访问权才能广播无效性操作。如果两个处理器同时对一个共享块进行写操作时，其广播无效操作的请求会通过总线的仲裁实现串行化。

第一个取得总线控制权的处理器将把另一个处理器的副本置为无效,这样写操作就会严格地串行执行。要使用这个方案对共享数据项执行写操作,首先要取得总线控制权,否则无法完成。所有的一致性方案都要求通过某种方式来实现对同一Cache块的串行化访问,不论是通过串行化对通信媒介的访问,还是通过串行化对其他共享结构的访问。

除了要使执行写操作的Cache数据块的副本无效外,还需要在Cache缺失时对数据项进行定位。在写直达Cache中,查找数据项的最新值非常容易,因为写入的数据总要送到存储器,故而可以在存储器中找到某个数据项的最新值(写缓冲区引起的额外复杂性会在下一章中讨论)。在设计时,如果有足够的存储器带宽来支持处理器的写操作,使用写直达法可以简化Cache一致性的实现。

对于写回式Cache来说,要找出数据最新的值比较困难,因为数据项的最新值可能不在存储器中而在Cache中。幸运的是,写回式Cache能够使用同样的监听方案来处理Cache缺失和写操作:所有处理器都要监听总线上的每个地址。如果一个处理器发现它留有被请求的Cache块的一个脏副本,它会对读操作做出响应,提供这个Cache块,并中断对存储器的访问。额外的复杂性来自于从处理器Cache中重新找回Cache块,如果各处理器是在彼此独立的芯片上,那么这个时间通常要比从共享存储器中找回Cache块的时间长。因为写回式Cache对存储器带宽的要求较低,而且支持更多更快的处理器,因此虽然它比较复杂,但多处理器系统还是倾向于使用这种方案。因此,我们主要讨论使用写回式Cache的实现机制。

可以利用Cache中已有的标识位来实现监听过程。其中数据块的有效位使得实现无效操作更加容易。无论是由无效操作还是由其他事件导致的读操作缺失,都可以通过监听总线来直接解决。对于写操作,需要知道其他Cache中是否存有该数据块的副本。这是因为如果没有其他副本,则在写回式Cache中就不用将写操作放到总线上了。不发送写操作能够节省时间和带宽。

要跟踪一个Cache数据块是否能被共享,需要为每一个数据块再增加一个状态位,就像有效位和脏数据位一样。增加了标志该数据块是否被共享的状态位之后,执行写操作时就可以据此来判定是否需要发送无效操作。当对共享的数据块执行写操作时,Cache会在总线上发送一个无效操作并且把该块标记为私有。之后,处理器就不会再发送该数据块的无效操作了。拥有Cache块唯一副本的处理器通常称为该Cache块的所有者。

当发送无效操作时,所有者的Cache块从共享状态变成非共享(或独占)状态。如果稍后其他处理器请求这个Cache块,那么它会再次变成共享状态。因为Cache对总线的监听同样可以观测到缺失的现象,所以它能够发现其他处理器请求独占Cache块的情况,并且会把状态设置成共享。

因为每个总线事务都必须检查Cache地址的标识,这就有可能干扰处理器对Cache的访问。一种减少这种干扰的方法就是复制标识,在多级Cache中还有一种方法就是将监听请求转给二级Cache。处理器在一级Cache发生缺失时才使用二级Cache。要实现这种方案,一级Cache的每个条目必须在二级Cache中呈现,这种特性称为包含性。如果监听在二级Cache命中,它就必须判定一级Cache应更新状态,而且可能还要一级Cache重新得到数据,这些通常会要求处理器停顿。在一些情况下将标识复制给二级Cache是十分有用的,这可以进一步减少处理器和监听行为之间的竞争。我们会在下一章更详细地讨论包含性。

协议范例

监听一致性协议一般通过每个节点的有限状态控制器来实现。这个控制器对每个来自处理器和总线的请求做出响应,然后改变选定的Cache块的状态,并可以使用总线来访问数据或使之无效。在逻辑上可以认为单独的控制器是和每个块连在一起的;因此对不同块的监听操作或Cache请求可以分开进行。在实现时,单独的控制器允许对不同块的多操作以交叉存取的方式进行(即使是在

每次只允许一个Cache访问或总线访问的情况下,另一个操作也可以在一个操作完成前初始化)。虽然我们在下面的介绍中只提到总线,但是对任何一种互连网络,只要它们支持向所有一致性控制器及其所连接的Cache发送广播,都可以用来实现监听。

最简单的协议有三种状态:无效、共享和修改。共享状态是指块是有可能被共享的;修改状态是指块在Cache中已经被更新,需要注意的是修改状态暗示了此时的块已被独占。图4.5列出了节点中处理器-Cache模块产生的请求(图中的上半部分)以及来自总线的请求(图中的下半部分)。这种协议适用于写回式Cache,但也可适用于写直达Cache,只要将修改状态重新解释为独占状态并使用普通写直达Cache的写操作更新Cache就能实现。这种基本协议最常见的扩展是加入独占状态,表明块是未修改的并被单独一个Cache所独占。图4.5的解释文字详细描述了这种状态。

请求	来源	地址 Cache 块状态	Cache 操作 类型	功能和解释
读命中	处理器	共享或修改	一般命中	读 Cache 中的数据
读缺失	处理器	无效	一般缺失	在总线上发读缺失消息
读缺失	处理器	共享	替换	地址冲突缺失: 在总线上发读缺失消息
读缺失	处理器	修改	替换	地址冲突缺失: 写回阻塞, 在总线上发读缺失消息
写命中	处理器	修改	一般命中	在 Cache 中写数据
写命中	处理器	共享	一致	在总线上发写缺失消息
写缺失	处理器	无效	一般缺失	在总线上发写缺失消息
写缺失	处理器	共享	替换	地址冲突缺失: 在总线上发写缺失消息
写缺失	处理器	修改	替换	地址冲突缺失: 写回阻塞, 在总线上发写缺失消息
读缺失	总线	共享	无动作	允许存储器处理读缺失
读缺失	总线	修改	一致	尝试共享数据: 数据块挂到总线上, 并设为共享态
无效	总线	共享	一致	尝试写共享数据块; 使块无效
写缺失	总线	共享	一致	尝试写共享数据块; 使 Cache 块无效
写缺失	总线	修改	一致	尝试写在别处独占的块: 写回该块, 并设为无效态

图 4.5 Cache一致性机制接受来自处理器和总线的请求并根据请求的类型(是否在Cache中命中)和请求中指明的Cache块状态来做出响应。第四列描述了Cache状态,包括一般命中、一般缺失(同单处理器Cache)、替换(单处理器Cache替换缺失)和一致(被要求保持Cache一致性)一般或替换操作可能根据块在其他Cache中的状态而导致块状态的一致性操作。对于总线侦测到的读或写缺失,只有当读或写地址与Cache中的块地址一致并且此块有效时才会执行动作。一些协议还包括待定状态,即块在Cache中是独占但尚未被写入的状态。如果写访问被分为下面两个步骤,就会导致这种状态的出现:先独占Cache中的块,然后再更新。在这样的协议中,这种“独占的未修改状态”是短暂的,写操作完成后也就消失了。其他协议对未修改块使用和维持一个独占状态。在监听协议中,当处理器对不在其他任何Cache中的块进行读操作时,就会进入这种状态。因为所有的后续访问都是受到监听的,所以就有可能保持这种状态的准确性。特别是,当其他处理器发生读缺失时,该状态由独占转为共享。增加这种状态的好处是,同一个处理器对独占块进行后续的写操作时,不需要请求总线访问权或产生无效操作,因为在该Cache中已经知道该块是独占的了;处理器很少将这种状态变为修改状态。通过使用状态比特位表明一个块的独占状态,使用重写位表明一个块的修改状态,可以很容易地扩展到该状态。最流行的MESI协议(其名称即四个包含状态:修改、独占、共享、无效)使用的就是这种结构。MOESI协议做了扩展:加入了所有者状态(在图4.4的说明中提到)

如前所述,在处理器或总线的激励下,每个Cache只有一个有限状态机。图4.7显示的是图4.6右半部分的状态转换与左半部分相结合后形成的单个Cache数据块的状态图。

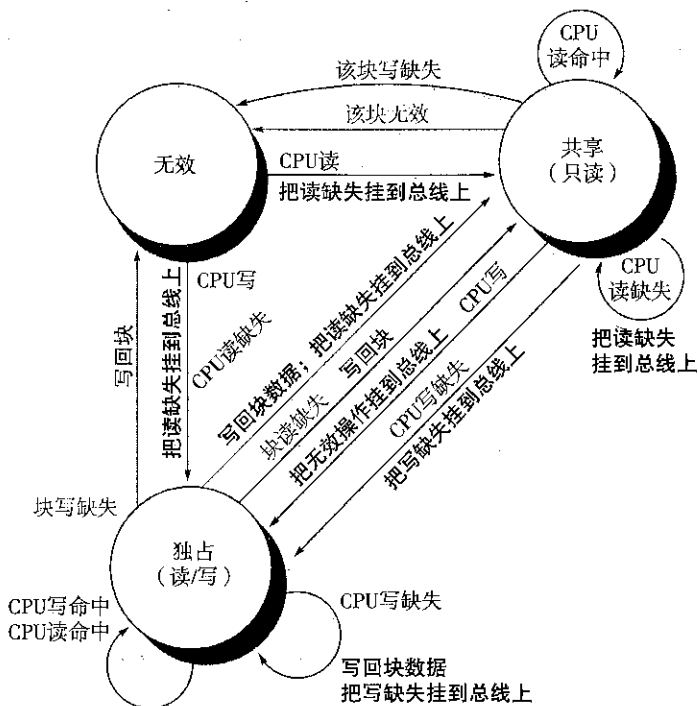


图4.7 Cache一致性状态图,由本地处理器激发的状态转换用黑体字表示,由总线活动激发的状态转换用灰体字表示。如图4.6所示,事务活动用粗体字表示

这个协议工作机理的关键在于任何一个Cache块要么在多个Cache中处于共享状态,要么在唯一一个Cache中处于独占状态。任何状态要转换到独占状态(处理器要对某个数据块执行写操作时必然会转换到这个状态),必须要在总线上放置一个写缺失信号,使所有的Cache把数据块置为无效。而且,如果其他Cache中含有处于独占状态的数据块,该Cache要通过执行写回操作来提供目标数据块。最后,如果在独占状态下数据块发生读缺失时,拥有该数据块的Cache也会使其状态变成共享,这样后续的写操作就能够获得独占权限。

图4.7中以灰体字表示的动作是该协议中的监听部分,由它们处理总线上读缺失和写缺失的情况。这个协议以及其他大部分协议的另一个特性就是,存储器中任何处于共享状态的数据块总是最新的。这个特性简化了实现方法。

虽然这个简单的Cache协议是正确的,但是它忽略了诸多复杂因素,而这些因素会使得实际的实现更加困难。其中最重要的是协议假设是原子操作——就是说,一个操作可以在没有其他操作干扰的情况下完成。例如,上面所描述的协议假设检测写缺失,获取总线以及接受应答是单一的原子操作。在实际操作中这种情况是不可能存在的。例如,如果使用一个交换机,就像最近所有的多处理器那样,甚至连读缺失也不会是原子操作。

非原子操作可能会造成协议死锁,即进入了一个无法继续执行的状态。我们要探究这些协议怎样才能没有总线的情况下被立即实现出来。

构建小规模(2至4个处理器)的基于总线的多处理器已经变得很容易。例如,Intel Pentium 4 Xeon和AMD Opteron处理器都被设计成可用于Cache一致性多处理器系统,而且有支持监听的外

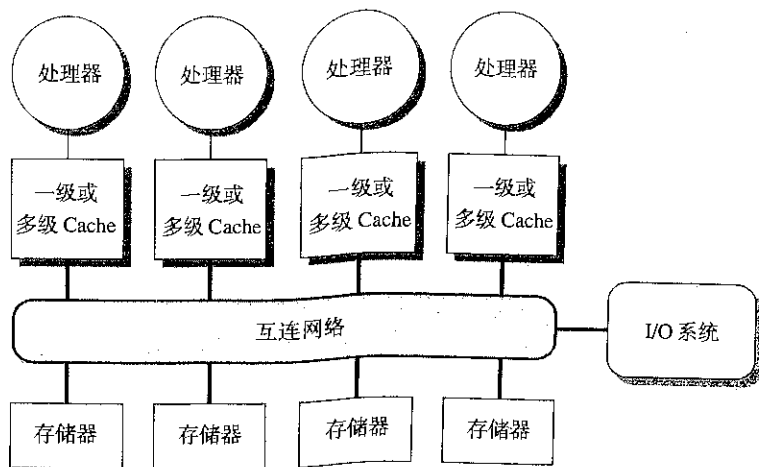
部接口,此接口还可直接连接2至4个处理器。为了减少对总线的使用,它们都有很大的片内Cache。而在Opteron处理器的例子中,对互连的多处理器的支持技术被集成到处理器芯片上,就像存储器接口一样。在Intel处理器的设计中,双处理器系统可以构建在只有一些额外扩展的芯片上,通过扩展的接口访问存储器系统和I/O。虽然这些设计不能被很容易地扩展到更大处理器数目的设计中,但是它们对于2至4个处理器系统提供了具有成本优势的解决方案。

下一节将研究这些协议用于并行和多道程序工作负载时的性能,通过研究性能,将会清楚地看到这些对简单协议进行扩展的价值。但是在此之前,先要简要地看一下使用对称式共享存储器系统结构和监听一致性设计的局限性。

对称式共享存储器多处理器和监听协议的局限性

随着多处理器中处理器数目的增加,或是处理器对存储器要求的增加,系统的任何集中式资源都会变成“瓶颈”。在最简单的基于总线的多处理器的例子中,总线必须同时支持一致性通信和由于Cache导致的存储器通信。同样,如果只有一个存储器部件,它就必须处理所有的处理器请求。随着处理器在过去几年的飞速发展,单独一条总线或单独一个物理存储器所能支持的处理器数量正在下降。

设计者怎样才能增加存储器的带宽以支持更多或更快的处理器?为了增加处理器和存储器之间的通信带宽,设计者使用多种总线以及各种互连网络,像交叉开关或小型点对点网络。在这样的设计中,存储器系统可以被配置成多个物理组,以便在保持均匀存储器访问时间的前提下增加有效存储器带宽。图4.8给出了这种方法,这正是本章开始时讨论的两种方法(集中式共享存储器和分布式共享存储器)的结合。



AMD Opteron代表了介于监听协议和目录协议之间的另外一种方法。存储器直接连到每个双核处理器芯片上,总共有4个处理器和8个内核。Opteron通过使用点对点连接向其他三个处理器广播的方法实现一致性协议。因为处理器内部的连接不是共享的,处理器得知一个无效操作完成的唯一途径只能是显式确认的方法。因此,一致性协议使用广播来找到可能的共享副本(类似于监听协议),但是使用确认来安排操作(类似于目录协议)。有意思的是,远程存储器时延和本地存储器时延相差并不太大,这样操作系统就认为Opteron多处理器具有均匀的存储器访问。

使用监听 Cache 一致性协议可以不要求拥有集中式总线,但仍然要求完成广播,以监听单独的 Cache,获取每个可能的共享存储器块的缺失。这种 Cache 一致性通信限制了处理器的扩展和速度。因为一致性通信量和 Cache 容量没有关系,更快的处理器相对于和 Cache 响应其他 Cache 监听请求的能力来说就显得能力过剩了。在 4.4 节中,我们将会研究目录协议,它不需要在发生缺失时向所有 Cache 进行广播。随着处理器速度和每个处理器内核数量的增加,越来越多的设计者倾向于选择这种协议来避免监听协议在广播上的局限性。

实现监听 Cache 一致性

细节决定成败。

经典谚语

1990 年本书的第一版完成时,在最后的综合一章介绍的是使用基于监听一致性的 30 个处理器、单总线多处理器;该总线容量才刚过 50 MB/s,这样的总线带宽甚至无法支持单个 2006 年生产的 Pentium 4 处理器!当 1995 年本书的第二版完成时,第一个使用多条总线的 Cache 一致性多处理器才刚刚出现,我们在附录中加入了介绍使用多总线在系统中实现监听的内容。2006 年,每个拥有多于两个处理器的多处理器系统都使用互连网络而不是单总线,设计者必须面对在无法将总线简化为串行化事件的情况下,实现监听所带来的挑战。

就像在前面所提到的,实现监听一致性协议的主要复杂性是在任何最新的多处理器中,写缺失或更新缺失都不是原子操作。监测写缺失或更新缺失、与其他处理器和存储器的通信、对于写缺失得到最新值、保证每个无效操作都被处理以及更新 Cache,这些步骤均无法在一个单循环中完成。

在简单的单总线系统中,可以通过首先竞争总线(在改变 Cache 状态之前)和直到操作完成才释放总线的方式将这些步骤变成有效的原子操作。处理器如何知道所有的无效操作均已完成?在大多数基于总线的多处理器中,当所有的无效操作被收到和处理时,会用单独的信号线来发出信号。信号发出后,产生缺失的处理器就可以释放总线,因为它已经知道在与下一次缺失相关的任何活动之前所有必须的操作都将完成。在上述步骤中,通过独占总线,处理器就可以有效地将这些单独的步骤变成原子操作。

在没有总线的系统中,必须找到发生缺失时将 these 步骤变为原子操作的其他方法。尤其是,必须保证两个处理器试图同时对相同的块进行写操作时(即称为“竞争”的情况),写操作能够被严格排序:一个写操作必须在另一个写操作开始前完成。只要保证竞争只有一个胜利者且其一致性操作首先完成即可,至于竞争的两个写操作谁先执行是无关紧要的。在监听系统中,保证这种竞争只有一个胜利者的方法是,对所有的缺失使用广播或者通过互连网络的一些基本特性来实现。这些特性以及重新启动竞争失败者处理缺失的能力,是在没有总线的情况下实现监听 Cache 一致性的关键所在。我们在附录 H 中会介绍更多的细节。

4.3 对称式共享存储器多处理器的性能

在一个使用监听一致性协议的多处理器中,几个不同的现象共同决定系统性能。特别是 Cache 的总体性能由单处理器的 Cache 缺失通信和由通信造成的数据传输共同决定。后者会造成无效和随之而来的 Cache 缺失。改变处理器个数、Cache 大小、块大小都会以不同的方式影响这两种缺失率,导致系统整体性能受这两个因素的共同影响。

在附录 C 中,把单处理器缺失率分为三种(3C: capacity, 容量; compulsory, 强制; conflict, 冲突),并深入分析了应用程序表现以及可以改进 Cache 的方法。与之类似,由处理器间通信引起的缺失,通常称为一致性缺失,它可以分为两种。

第一种是所谓的真共享缺失,由 Cache 一致性机制下的数据通信产生。在基于无效的协议中,处理器对共享 Cache 块的第一次写操作引发无效事件,从而获得该块的独占权。除此之外,另一个处理器试图从这个 Cache 块中读一个已修改的字时,会产生一个缺失并且传送一个结果块。因为这两种缺失都直接由处理器间共享数据引起,所以都被归为真共享缺失。

第二种影响,即假共享,是由于使用基于无效的一致性算法引起的。这种算法利用了数据块的有效位,而这个有效位每个数据块只有一个。对数据块中的某些字执行写操作而导致该数据块被置为无效后(随后对该块的调用会导致缺失),再对该块中另外一些字执行写操作时,就会发生假共享现象。如果在写操作中将其他处理器中的数据副本置为无效之后,这些处理器再次使用已经写入的数据,那么该调用是一个真共享调用并会导致缺失,而缺失与块大小或字的位置无关。然而,如果写入的字和读取的字不相同,那么无效并不会引发传输新数值的操作,而只是导致多余的 Cache 缺失,这就是一个假共享缺失。在假共享缺失中,共享的是数据块,但是并没有共享单个的字,如果数据块大小是一个字,则不会发生这种缺失。下面的例题详细解释了各种共享方式。

例题 假设字 x1 和 x2 处于同一块 Cache 中,这个 Cache 块在 Cache P1 和 P2 中均为共享状态。P1 和 P2 两个 Cache 在这之前已经读入了 x1 和 x2。假设有如下事件序列,请区分每个缺失究竟是真共享 Cache,还是假共享 Cache,或是命中。如果数据块大小是一个字,任何能够发生的缺失就都是真共享缺失了。

时序	P1	P2
1	写 x1	
2		读 x2
3	写 x1	
4		写 x2
5	读 x2	

解答:按事件顺序给出:

1. 是真共享缺失。因为 P2 已经读取了 x1, 需要将 P2 上的 x1 设置为无效。
2. 是假共享缺失。因为 x2 是由于 P1 对 x1 执行写操作而置成无效的, 但 P2 中并没有使用 x1 的值。
3. 是假共享缺失。因为数据块是由于在 P2 中读 x2 而被标记为共享的, 但 P2 并没有读取 x1。包含 x1 的 Cache 块在 P2 读后被标记为共享状态, 而写缺失需要获得块的独占访问权。在某些协议中, 可以通过更新请求解决这个问题, 即只产生一个总线无效, 但是并不传输该 Cache 块。
4. 是假共享缺失。理由与 3 相同。
5. 是真共享缺失。因为要读取的值刚被 P2 写过。

虽然可以看到商业负载中真假共享缺失的影响,但对于紧耦合的应用来说,一致性缺失的影响更大。我们在附录 H 中考虑并行科学计算的性能时,会详细讨论它们的影响。

商业负载

本节研究四处理器共享存储器多处理器的存储器系统的行为。数据取自 AlphaServer 4100 或是一个 AlphaServer 4100 的可配置仿真器。AlphaServer 4100 的每个处理器都是 Alpha 21164, 其在

每个时钟周期可以发射4条指令,以300 MHz的速度运行。虽然系统中的处理器时钟周期比近期其他系统的处理器要慢很多,但是系统的基本架构,包括四发射处理器和三级Cache系统结构都与最新的系统很相似。特别是,每个处理器都有三级Cache系统结构:

- 一级Cache包括一对8 KB的直接映射片内Cache,一个用于存放指令,另一个用于存放数据。Cache块大小为32字节,数据Cache用一个写缓冲区对二级Cache写直达。
- 二级Cache是一个96 KB三路组相联内置Cache,块大小为32字节,使用写回方式。
- 三级Cache是2 MB直接映射的外部Cache,块大小为64字节,使用写回方式。

对二级Cache的访问时延为7个时钟周期,三级Cache为21个时钟周期,主存时延为80个时钟周期(一般没有竞争)。由另一块Cache中的独占数据的缺失而产生的Cache对Cache的数据传送需要125个时钟周期。既然这些缺失造成的代价比目前的高时钟周期系统的缺失代价要小,而且Cache相对也更小,这也就意味着目前的系统会有更低的缺失率,但同时也有更大的缺失代价。

用于该研究的负载由三部分应用构成:

1. 仿照TPC-B的联机事务处理负载(OLTP)模型,并使用Oracle 7.3.2作为底层的数据库,其中TPC-B和其最新的版本TPC-C拥有相似的存储器行为。这种负载由一个产生请求的客户进程集合及处理这些请求的服务器进程集合构成。其中服务器进程占据了85%的用户时间,剩余的属于客户进程。虽然通过仔细的调度和足够的请求可以保持处理器的忙状态,从而掩盖I/O的时延,但是需要I/O操作的服务器进程通常会在25 000条指令后发生阻塞。
2. 基于TPC-D的决策支持系统(DSS)负载,同样使用Oracle 7.3.2作为底层的数据库。尽管该负载只包括TPC-D中17条读查询中的6条,虽然这6条查询涵盖了整个基准测试程序的所有活动。为了掩盖I/O的时延,在查询中和查询间都要实现并行机制,对于前者,是在查询中是在查询形成的过程中完成并行的。这里的阻塞情况没有OLTP中那么频繁,6条查询平均下来要1 500 000条指令才发生一次阻塞。
3. 基于AltaVista数据库(200 GB)存储器映射版本查询的Web索引查询(AltaVista)基准测试程序。内循环得到了很好的优化。因为查询结构是静态的,线程间几乎不需要同步。

用户模式、内核模式和空闲循环所占的时间比例见图4.9。I/O的频率同时提高了内核时间和空闲时间(从OLTP行可以看出,它有最大的I/O和计算比率)。AltaVista将整个查询数据库映射到存储器,并且经过了大范围的调整优化,因此拥有最小的内核或空闲时间。

基准测试程序	用户模式时间百分比	内核时间百分比	CPU 空闲时间百分比
OLTP	71	18	11
DSS (所有查询的平均访问)	87	4	9
AltaVista	>98	<1	<1

图4.9 商业负载中执行时间的分布图。OLTP基准测试程序拥有最大的操作系统时间和CPU空闲时间(即I/O等待时间)。DSS基准测试程序的操作系统时间和I/O活动都要少一些,但空闲时间仍然有9%。可以看出AltaVista搜索引擎的大范围调整在这些测试中效果显著。数据由Barroso等人在四处理器AlphaServer 4100上采集得到[1998]

商业负载的性能测试

从观察基准测试程序在这个四处理器系统上的执行情况开始,正如在前面一小节讨论的,这些基准测试程序包括大量的I/O时间,这在CPU时间测试中是被忽略的。我们把6个DSS查询语句组

成一个单独的基准测试程序,来测试平均性能。这些测试程序的有效CPI各不相同,AltaVista网页搜索的CPI为1.3,DSS负载的平均CPI为1.6,OLTP负载为7.0。图4.10说明了执行时间如何分成指令执行、Cache和存储器系统访问时间以及其他的时延(其他时延主要是指流水线资源时延,但也包括TLB和转移预测错误的时延)。虽然DSS和AltaVista负载的性能尚可,但OLTP负载的性能非常差,这是由于存储器体系的性能不理想造成的。

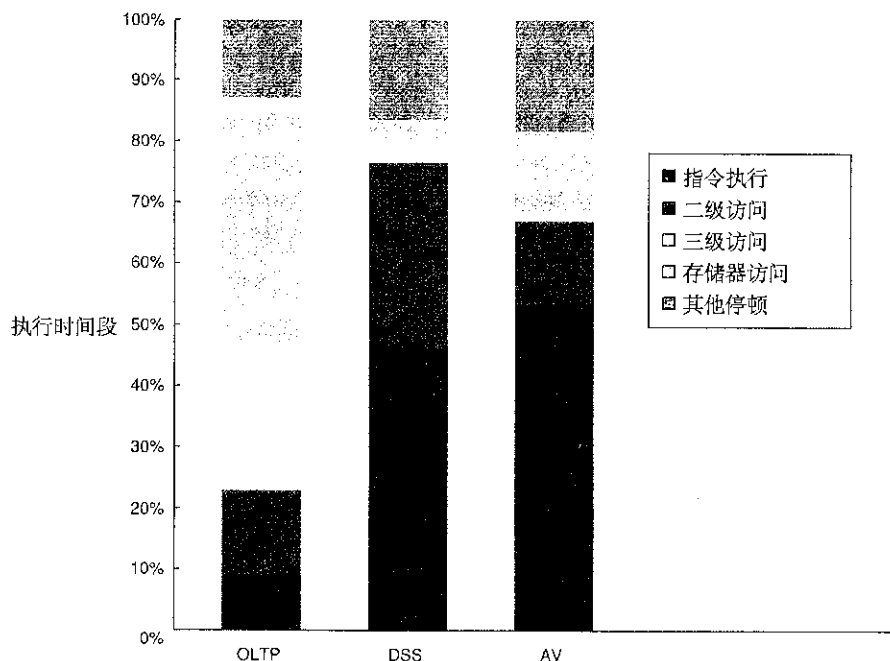


图4.10 商业负载中三种程序的执行时间分类(OLTP, DSS和AltaVista)。DSS的数据是6种不同的查询语句的平均值。CPI分别是AltaVista的1.3, DSS的1.61, Oracle的7.0(DSS单独查询时CPI的变化范围是1.3~1.9)。其他的延迟包括:资源延迟(在21164使用重放陷阱实现),转移预测错误,存储器阻塞,TLB缺失。对这种基准测试程序,基于资源的流水线延迟是主要因素。这个数据综合了用户行为和系统内核访问。只有OLTP有明显的内核访问,而且内核访问比用户访问的性能表现要好。本节所有的测试数据都是由Barroso, Gharachorloo和Bugnion搜集的[1998]

由于OLTP负载访问存储器最频繁,也有很多的三级Cache缺失,我们集中考察三级Cache大小、处理器个数和Cache块大小对OLTP基准测试程序的影响。图4.11说明了使用两路组相联Cache所产生的影响,增加的Cache大大减少了冲突缺失。增大三级Cache减少了三级缺失,因此执行时间得到改进。令人惊讶的是,几乎所有的时间改进都是在三级Cache从1MB增加到2MB时产生的,若继续增加其容量,虽然Cache缺失对于2MB和4MB的Cache仍然是性能损失的主要原因,但在时间上的改进已几乎不可能。这是为什么?

为了理解和回答这个问题,我们必须确定哪些因素影响三级Cache缺失率,以及当增加三级Cache容量时这些因素的变化情况。图4.12从五个方面说明了每个指令所带来的存储器访问周期数。1MB三级Cache两个最大的存储器访问周期来源是指令和容量/冲突缺失。随着三级Cache的增大这两个方面变得不太重要。遗憾的是,增大三级Cache对强制、假共享和真共享缺失则不会产生影响。所以,对于4MB和8MB的Cache,真共享缺失是缺失中最大的部分;因此当三级Cache容量超过2MB时,增大Cache容量不会减小真共享缺失,因此总的缺失率的减少也就很有限了。

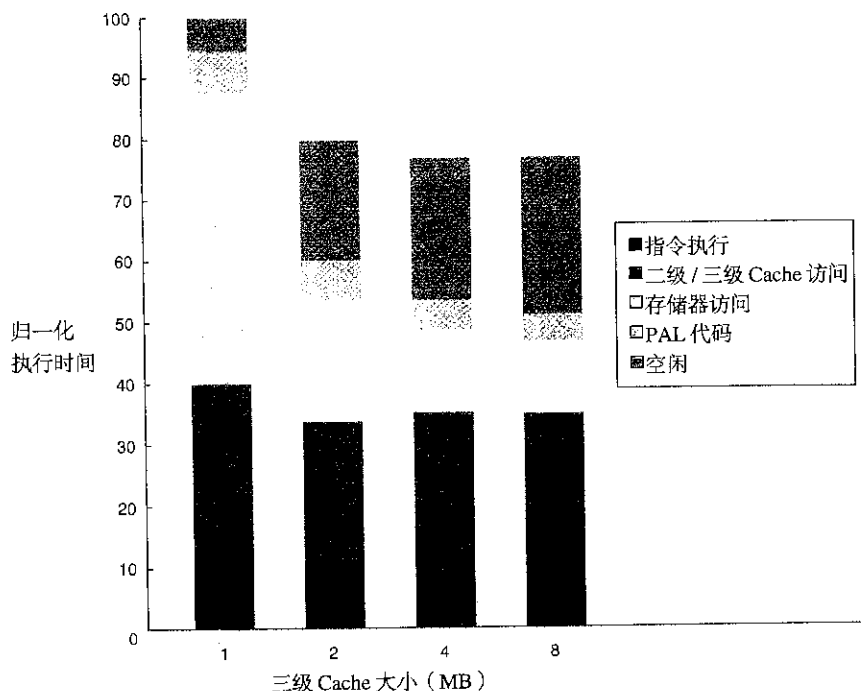


图 4.11 当二路组相联三级 Cache 从 1 MB 增加到 8 MB 时 OLTP 负载的相对性能。随着 Cache 容量的增加，空闲时间也随之增加，这就降低了一部分性能改善的程度。造成这种现象的原因是由于存储器时延的减少，需要更多的服务器进程来掩盖 I/O 时延。可以通过重新调整负载来增加计算 / 通信平衡，并控制空闲时间

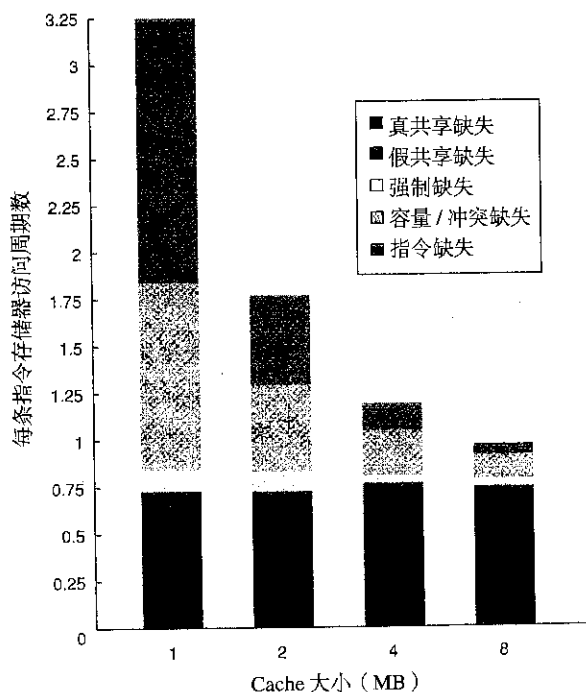


图 4.12 当 Cache 大小增加时各种因素对存储器访问周期的影响。假定三级 Cache 为二路组相联

增大 Cache 消除了绝大多数单处理器的缺失, 但并未对多处理器缺失产生影响。那么, 增加处理器个数对各种类型的缺失将产生什么样的影响呢? 图 4.13 列出了基于二路组相联的 2 MB 三级 Cache 的数据。正如所期望的那样, 真共享缺失率的增加 (并未受单处理器缺失率下降的影响) 导致每条指令中访存周期的增加。

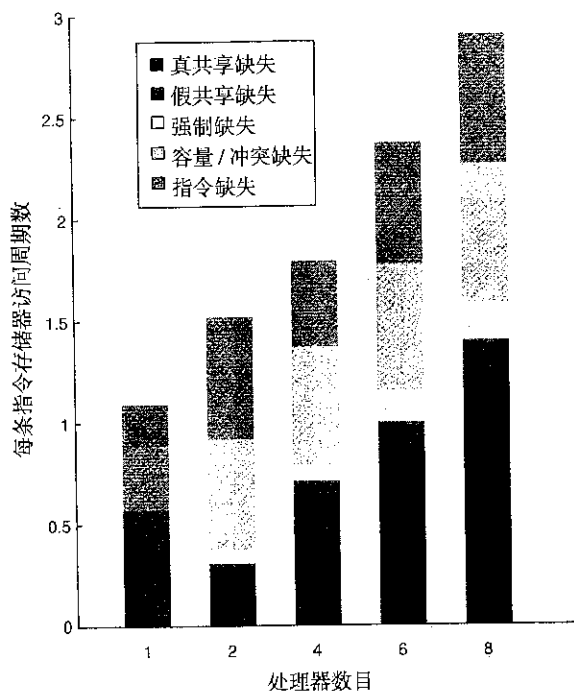


图 4.13 当处理器数增加时真共享对访存周期的影响。因为每个处理器现在必须处理更多强制缺失, 所以强制缺失有了轻微的增加

最后一个问题是增加 Cache 块大小 (可能减少指令和强制缺失率, 并在有限范围内也降低容量或冲突缺失率) 是否对此负载有所帮助。图 4.14 显示了当块大小从 32 字节增加到 256 字节时每 1000 条指令的缺失率。把块大小从 32 字节增加到 256 字节对以下缺失率的四个因素产生了影响:

- 真共享缺失率降至不到原来的 1/2, 这说明了真共享的局部性。
- 正如所预期的那样, 强制缺失率显著降低。
- 冲突/容量缺失有一定减少 (块大小增长 8 倍时缺失率降低 1.26 倍), 说明空间局部性在单处理器缺失 (使用大于 2 MB 的三级 Cache) 中并不显著。
- 假共享缺失率虽然绝对数量并不大, 但几乎增加了一倍。

指令缺失率没有受到显著影响这一事实令人感到意外。如果一个单指令 Cache 有这样的表现, 可以断定空间局部性是很差的。对于混合型二级 Cache 而言, 诸如指令-数据竞争的影响同样也会导致较大块的指令 Cache 的高缺失率。其他研究已经表明在较大数据库和 OLTP 负载 (它们有很多小的基本块和专用的代码顺序) 的指令流中, 空间局部性是很差的。尽管如此, 把三级 Cache 的块大小增加到 128 字节或 256 字节看起来也更为合适。

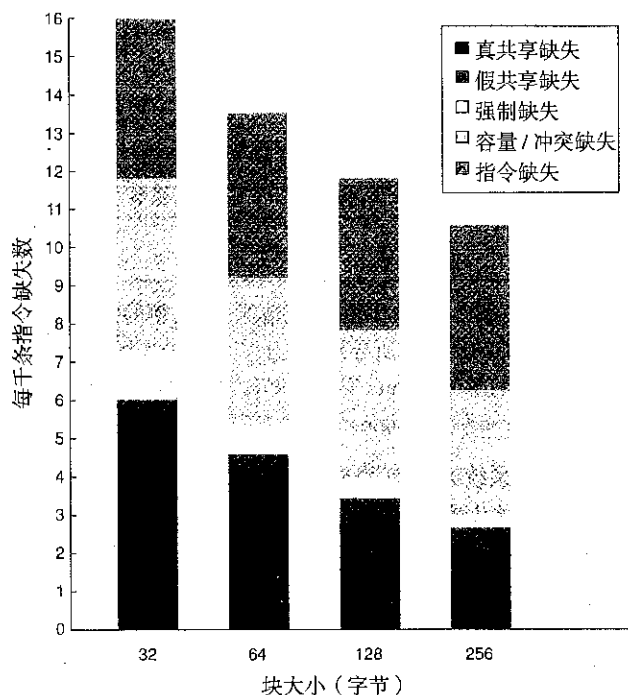


图 4.14 每千条指令的缺失数随三级 Cache 块大小的增加平缓下降，在块大小至少为 128 字节时情况很好。三级 Cache 为 2 MB 的二路组相联

多道程序和操作系统负载

下面要研究的内容是由用户活动和操作系统活动构成的多道程序负载。负载选用 Andrew 基准测试程序编译阶段的两个独立副本，它是仿真软件开发环境的基准测试程序。该编译阶段由 8 个处理器并行完成。负载在八处理器上运行 5.24 秒，产生 203 个进程并执行对 3 个不同文件系统的 787 次磁盘请求。负载使用 128 MB 存储器，且无页面调度活动的发生。

负载分为 3 个不同阶段：编译基准测试程序（其中包括子计算活动）；在库里建立对象文件；移除对象文件。最后一个阶段完全由 I/O 控制而且只有两个活动的进程（每个负责一个运行）。在中间阶段，主要是依靠 I/O，处理器大部分时间是空闲的。整个负载中，系统活动和 I/O 活动比起经过大调整的商业负载来说要更加密集。

对于负载测试来说，我们对存储器和 I/O 系统做如下假设：

- 一级指令 Cache: 32 KB，两路组相联，块大小为 64 字节，命中时间为 1 个时钟周期。
- 一级数据 Cache: 32 KB，两路组相联，块大小为 32 字节，命中时间为 1 个时钟周期。我们会改变一级数据 Cache 以探究它对 Cache 行为的影响。
- 二级 Cache: 1 MB，指令和数据共用，两路组相联，块大小为 128 字节，命中时间为 10 个时钟周期。
- 存储器：挂在总线上的单存储器，访问时间为 100 个时钟周期。
- 磁盘系统：固定访问时延为 3 ms（比正常的要短，为了减小空闲时间）。

图 4.15 给出了怎样使用所列出的参数将执行时间划分到 8 个处理器中。执行时间由四部分组成：

1. 空闲：在内核模式的空闲循环下执行。
2. 用户：在用户模式下执行。
3. 同步：执行或等待同步变量。
4. 内核：在非空闲或同步访问模式的操作系统中执行。

	用户执行	内核执行	同步等待	CPU 空闲 (等待 I/O)
执行指令的比例	27	3	1	69
执行时间的比例	27	7	2	64

图 4.15 在多道程序并行构造负载中执行时间的分配情况。高比例的空闲时间是因为 8 个处理器中只有一个活动时磁盘的空闲时间所导致。关于该负载的本图数据和后面的测试数据都是在 SimOS 系统上收集的[Rosenblum 等, 1995]。实际的运行和数据收集由斯坦福大学的 M. Rosenblum, S. Herrod 和 E. Bugnion 完成

多道程序负载至少对于操作系统来讲有明显的指令 Cache 性能损失。在操作系统中, 64 字节块容量、两路组相联的指令 Cache 缺失率从 32 KB Cache 的 1.7% 变化到 256 KB Cache 的 0.2%。不论 Cache 的大小是多少, 用户级指令 Cache 缺失大约为操作系统速率的 1/6。这可以部分地解释虽然用户模式下执行的指令数为内核模式下的 9 倍, 但是前者的执行时间只是后者的 4 倍。

多道程序和操作系统负载的性能

这一小节研究改变 Cache 容量和块大小对多道程序工作负载的 Cache 性能产生的影响。内核和用户进程的行为存在差异, 因而对这两部分分别考察。但是由于用户进程执行的指令数要多 8 倍, 因此总的缺失率主要取决于用户代码的缺失率, 关于这一点, 下面会看到, 用户进程的缺失率经常是内核缺失率的 1/5。

虽然用户模式执行更多的指令, 操作系统的行为比用户进程会导致更多的 Cache 缺失。除了操作系统有更大的代码容量和局限性缺失这两个原因之外, 还有另外两个原因: 第一, 内核把页分配给用户之前首先要初始化所有页, 这极大地增加了内核的强制性缺失率; 第二, 内核实际上共享数据, 因此不能忽视一致性缺失率。相反, 对于用户进程来说, 只有被调度到另一个处理器上才会引起一致性缺失, 这一部分所占比例其实很小。

图 4.16 分别给出了数据缺失率与数据 Cache 大小的比率, 以及它对于内核和用户部分块大小的比率。可以看出增加数据 Cache 的大小对用户缺失率的影响比对内核缺失率的影响要大。增加数据块大小对缺失率有好的效果, 因为大部分缺失属于强制性缺失和容量缺失, 这两部分缺失都能通过增大数据块大小得到改进。因为一致性缺失部分所占的比例较小, 所以增加块大小所产生的负面效应是有限的。为了说明内核和用户进程行为差异如此之大的原因, 可以分析一下内核缺失的情况。

图 4.17 分别给出了随着 Cache 容量或块大小的增加内核缺失是如何变化的。缺失可以分成三类: 强制缺失, 一致性缺失 (源自真共享和假共享), 容量/冲突缺失 (包括操作系统和单个或多个用户进程之间的冲突所引起的缺失)。图 4.17 证实了对于内核调用来说, 增大 Cache 容量的大小只能减少单处理器的容量/冲突缺失率。与此相反, 增大块大小可以减少强制缺失率。随着块大小的增加, 一致性缺失率并没有增长很多, 这表明假共享的影响是无关紧要的, 虽然这种缺失会抵消一部分由于减少真共享缺失而带来的性能增加。

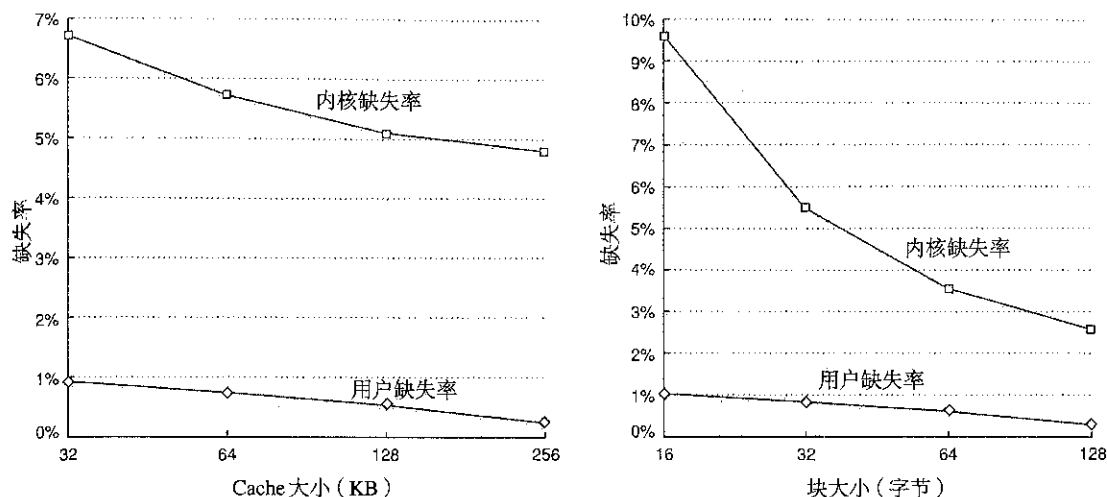


图 4.16 增加一级数据 Cache 容量 (左图) 或一级数据 Cache 块大小 (右图) 时, 用户代码和内核代码的数据缺失率变化情况是不同的。Cache 数据块为 32 字节时, 当数据 Cache 从 32 KB 增加到 256 KB 时, 用户代码的数据缺失率要比内核代码下降得快。用户层缺失率减少了 2/3, 内核层的缺失率只减少为最初的 1/3。在一个具有 32 KB 的两路组相联数据 Cache 和 8 个处理器的多处理器中, 增大块大小时两者的缺失率稳步减少。但与增大数据 Cache 容量不同的是, 增加数据 Cache 块大小对内核调用缺失率的改善更显著 (块大小从 16 字节增加到 128 字节的过程中, 内核引用几乎减少了 3/4, 而用户引用仅仅减少了 2/3)

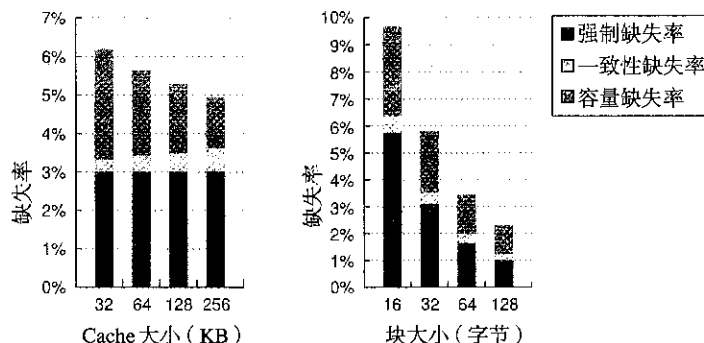


图 4.17 当多道程序工作负载运行在 8 个处理器上时, Cache 容量从 32 KB 增加到 256 KB 时内核数据缺失率各组成部分的变化。强制缺失率部分保持不变, 因为它不受 Cache 容量的影响。容量缺失率至少减少了 1/2, 而一致性缺失率几乎增加了一倍。一致性缺失增加的原因是, 由无效引起的缺失随 Cache 容量的增大而增加, 这是因为由于容量的关系而发生冲突的数据项越来越少了。正如我们期望的, 增加一级数据 Cache 块大小大大降低了内核调用中的强制缺失率。同时, 它也对容量缺失率有着很大的影响, 在一定块容量变化的范围内, 容量缺失率降低了 2.4 倍, 而增大块容量的大小对一致性通信的影响不大, 当块增加到 64 字节时一致性通信就稳定不变了, 而且即使增大到 128 字节一致性缺失率也几乎没有变化。由于一致性缺失率并不随着块容量的增长而减少, 因此如图所示一致性缺失所占总缺失的比例就从大约 7% 增长到 15%

如果考察每个数据调用所涉及到的字节数目, 如图 4.18 所示, 会发现内核调用具有较高的通信率, 且伴随块大小的增加快速增长。这是很容易理解的: 当块大小从 16 字节增大到 128 字节时,

缺失率大约降低 3.7 倍，但是发生每个缺失所要传输的字节数量提高了 8 倍，所以总的缺失通信量大约增加了 2 倍。对于用户调用来说同样有这种现象，不过其开始的通信量很小，所以还不如内核调用这么明显。

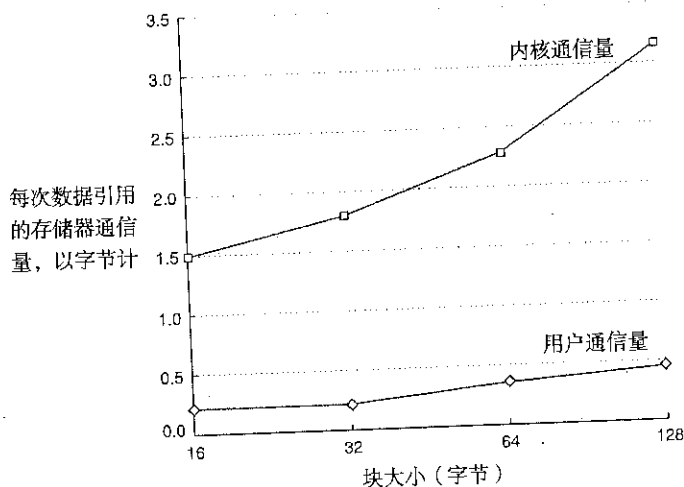


图 4.18 无论是程序的内核部分还是用户部分，每个数据调用需要的字节数随块大小的增加而增加。把该图同附录 H 中科学程序数据比较会得到有趣的结果

对于多道程序工作负载，操作系统对存储器系统的要求是非常苛刻的。如果工作负载中包含了更多的操作系统或类操作系统的活动，那么想要建造一个能力足够大的存储器系统是非常困难的。要改善性能也许可以通过更好的编程环境或编程帮助，使得操作系统更加关注 Cache 的活动。例如，对于来自于不同系统调用发出的请求，操作系统会重用存储器。虽然重用存储器将会导致完全的写覆盖情况发生，但是硬件并没有意识到这一点，而是继续试图保持 Cache 的一致性，并认为 Cache 块的一部分可能会被读取（虽然这不会发生）。这和过程调用中的栈位置重用情况类似。IBM Power 系列机允许编译器在程序调用时指明这种情况。但是在操作系统中很难发现这种行为，如果这样可能会要求程序员的帮助，但是这样做的代价可能更大。

4.4 分布式共享存储器和基于目录的一致性

就像我们在 4.2 节中所看到的，对于每个 Cache 缺失（包括对潜在共享数据的写操作），监听协议都需要和所有的 Cache 进行通信。监听协议中没有踪迹 Cache 状态的集中式数据结构，这一点从价格上来讲是个优点，但在需要扩展时就成了致命的弱点。

例如，在一台数据 Cache 为 512 KB、Cache 块大小为 64 字节的 16 个处理器的多处理器系统中，4 个科学/技术负载（附录 H）所要求的全部总线带宽（忽略了时延周期）需求在大约 4 GB/s 和 170 GB/s 之间。上述数据是在时钟周期为 4 GHz 的处理器上每 1 ns 执行 4 次数据调用的情况下得到的，这在 2006 年的超标量处理器没有 Cache 阻塞时可以做到。作为参照，2006 年最高性能的集中式共享存储器 16 路多处理器的存储器带宽为每个处理器 2.4 GB/s；而使用分布式存储器模型的多处理器从目前最新的存储器得到的最大可用带宽为每个处理器 12 GB/s。

因此，我们可以通过分配存储器的方式来增加存储器带宽和互连带宽，就像图 4.2 所说的那样，这种方式可以将本地存储器通信和远程存储器通信迅速分开，这就减少了对存储器系统和互连网络

的带宽要求。但是,除非能够消除一致性协议对每个Cache缺失都要广播的需求,否则分布式存储器也不会带来多少好处。

就像前面所提到的,基于监听的一致性协议的一种替代选择为目录协议。该协议保存每个Cache数据块的状态。目录中的信息包括哪个Cache拥有该块的副本,是否处于脏状态,等等。目录协议同样可以用于在集中式共享存储器中减少带宽需求,比如Sun T1的设计就是这样的(详见4.8节)。假设目录协议是在分布式存储器中实现的,但是相同设计也适用于按组进行组织的集中式存储器。

最简单的目录协议的实现机制是在目录中给每个存储器数据块分配一个条目。在这样的设计中,信息个数与存储器中块的个数(其中每个块的大小和二级或三级的Cache块大小一样)和处理器个数的乘积成正比。这对于处理器数目小于200的处理器系统来说并不成问题,因为目录的开销还可以接受。对于更大型的多处理器来说,就要想办法有效地扩展目录结构。已经提出的方法要么是减少需要保存信息的块数(例如,只保存Cache中数据块的信息而不是所有存储器数据块的信息),要么是使用单独的位数来代表包含少量处理器的集合,以减少每个条目中保存信息的数据块。

为了防止目录成为瓶颈,需要使目录随存储器分布,这样访问不同的目录就要到不同的地点,正如要在不同的存储器上执行不同的存储器请求一样。分布目录使得数据块的共享状态总是位于一个已知的地点。正是这个特性避免了一致性协议中的广播。图4.19显示了每个节点上带有目录的分布式存储器多处理器系统。

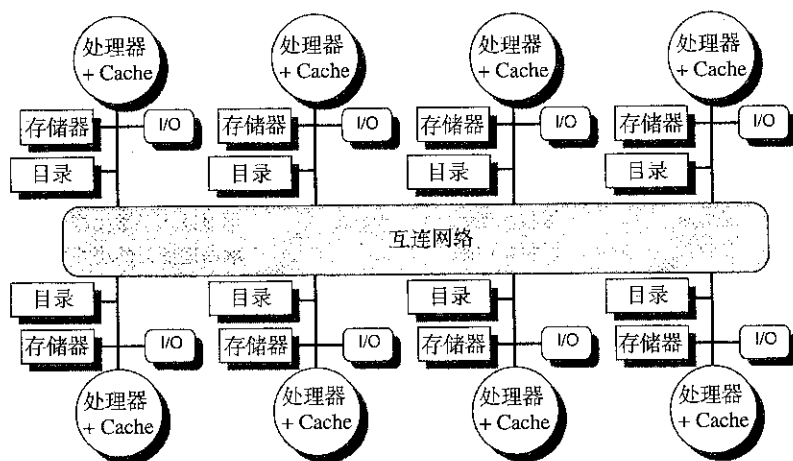


图4.19 在分布式存储器多处理器系统中,每个节点上用目录来实现Cache的一致性。每个目录负责跟踪共享本地存储器的Cache。目录可以按图中所示通过公用总线来与处理器和存储器通信,也可以通过专用端口连接到存储器,又或者可以作为中央节点控制器的一部分来实现。如果是作为控制器来实现,那么所有节点间和节点内的通信都要经过控制器

基于目录的Cache一致性协议：基础知识

正如监听协议一样,目录协议也必须实现两个基本操作:处理读缺失和处理共享未修改Cache块的写操作(处理共享数据块的写缺失由这两个操作简单组合而成)。为了实现这些操作,目录必须跟踪每个Cache块的状态。在一个简单的协议中,可能出现的状态如下:

- **共享**: 一个或多个处理器拥有Cache的数据块,并且存储器中的数值也是最新的(与所有的Cache中一样)。
- **未缓存**: 没有任何一个处理器含有该数据块的副本。

- **修改:** 只有一个处理器拥有该Cache块的副本并且对该块执行过写操作, 因此存储器中的副本是无效的。这个处理器称为该块的所有者。

除了跟踪Cache中每个块的状态外, 还必须跟踪拥有共享数据块副本的处理器, 因为执行写操作后这些处理器中的副本要设置成无效状态。实现这一功能的最简单方法是为每个存储器保留一个位向量。当块处于共享状态时, 向量的每一位表示所对应的处理器是否拥有该块的副本。当块处于独占状态时, 可以利用位向量来跟踪块的所有者。为了提高效率, 还要跟踪各个Cache中每个Cache块的状态。

每个Cache上的状态及转换与监听协议中的相同, 只是转换中执行的动作稍有不同。数据项的独占副本的无效化和定位过程是不同的, 因为两者都包括请求节点和目录间以及目录和一个或多个远程节点间的通信。在监听协议中, 这两步是通过向所有节点广播的方式结合在一起的。

在研究协议状态图之前, 先来考察一下为了处理缺失和保持一致性, 处理器和目录之间可能发送的消息类型。图4.20给出了在节点之间发送的消息类型。**本地节点**是指产生请求的节点。而**主节点**是指存储地址和目录条目所在的节点。物理地址空间是静态分配的, 因此给定一个物理地址, 就能知道该地址所对应的存储器和目录所在的节点。例如, 可以用高位表示节点号, 低位表示该节点存储器内的偏移地址。本地节点也可以是主节点。当主节点是本地节点时, 必须访问目录, 因为副本可能在第三个节点中存在, 这个节点叫**远程节点**。

消息类型	来源	目标	消息内容	消息的功能
读缺失	本地 Cache	主目录	P, A	处理器 P 在地址 A 缺失; 请求数据并将 P 设置为操作共享成员
写缺失	本地 Cache	主目录	P, A	处理器 P 在地址 A 写缺失; 请求数据并使用 P 成为独占所有者
无效	本地 Cache	主目录	A	向所有远程的 Cache 地址 A 块的 Cache 发送无效的请求
无效	主目录	远程 Cache	A	把地址 A 的数据副本设置为无效
取数据	主目录	远程 Cache	A	取回地址 A 的块并发送到它的主目录上; 把远程 Cache 中 A 的状态改为共享
取数据 / 无效	主目录	远程 Cache	A	取回地址 A 的块并发送到它的主目录上; 把 Cache 中的块置成无效
数据值应答	主目录	本地 Cache	D	从主存储器返回数值
数据写回	远程 Cache	主目录	A, D	写回地址 A 的数值

图 4.20 节点之间为保证一致性发送的可能消息, 包括源和目标节点, 消息内容 (P = 请求的处理器数目, A = 请求地址, D = 数据内容), 以及消息的功能。前 3 个消息是由本地节点发送到主节点的请求。第 4 个到第 6 个是当主节点遇到读缺失或写缺失时向远程节点发送的消息。主节点通过数据应答消息向请求节点发送数据。在两种情况下需要对数值执行写回操作: 一种情况是, 如果替换了 Cache 中的一个数据块, 且必须写回到它的主存储器中; 另一种情况是, 对来自主节点的取数据 / 无效消息做应答时。数据块一旦处于共享状态就执行写回操作, 这样做能简化协议中状态的数目, 因为任何脏数据块必须处于独占状态并且任何共享块总是可以在主存储器中获取

远程节点是指拥有 Cache 块副本的节点。该副本可处于共享状态, 也可处于独占状态 (这时是唯一的副本)。远程节点可以和本地节点或主节点相同。如果这样, 则基本协议不变, 但处理器间消息可能被处理器内消息代替。

本节中假设了一个存储器一致性的简单模型。为了减少消息类型和降低协议的复杂度, 我们假设消息能够按它们发送时的顺序被接收和执行。这个假设在实际中不一定正确, 并会导致其他复杂

的情况，我们会在4.6节中讨论存储器一致性模型时讨论其中的一些问题。本节用这个假设来保证处理器发出的无效操作能够在传输新的消息之前被优先考虑，就像在实现监听协议的讨论中所做的假设一样，忽略实现一致性协议的一些必要细节。尤其是，连续的写操作和对写操作无效性完成的确认并不像基于广播的监听机制那么简单。相反，对写缺失和无效操作的响应要求经过详细的确认。我们会在附录H中对此做进一步的讨论。

目录协议范例

目录协议中Cache块的基本状态与监听协议一样，而目录的状态也与前面列出的类似。因此首先研究简单的Cache块状态转换图，然后再研究带有目录条目的状态图。与监听的例子一样，这些状态转换图并没有把一致性协议的全部细节都表现出来；然而，实际的控制器在很大程度上取决于处理器的大量细节内容（消息传递性质，缓冲区结构，等等）。本节给出的是基本协议状态图，附录H中会列出一些在实现状态转换图过程中将碰到的棘手问题。

图4.21所示为单个Cache响应的协议动作。图中我们使用与上一节中相同的记号，来自外节点的请求用灰体字表示，而动作作用黑体字表示。单个Cache的状态转换是由读缺失、写缺失、无效和取数据引发的；这些操作在图4.21中都列出来了，单个Cache还会向主目录发送它所生成的读缺失消息、写缺失消息以及无效消息。读和写缺失要求有数据值应答，而且这些事件的状态会等到接收应答后才发生改变。确认无效操作何时完成是一个单独的问题，而且是单独处理的。

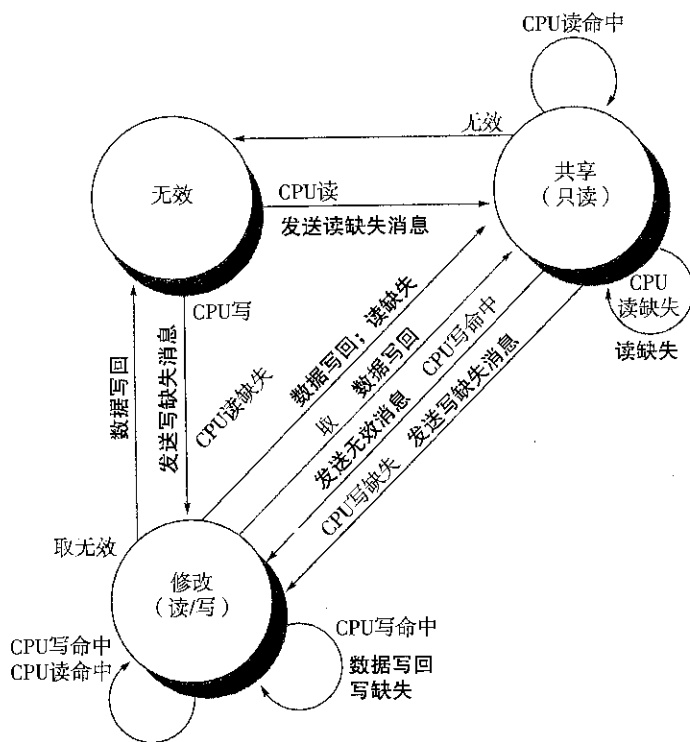


图4.21 基于目录的系统中单个Cache状态转换图。本地处理器的请求用黑体字显示，主目录的请求用灰体字显示。这些状态与监听情形下是相同的，事务处理也非常相似。不同之处在于执行写缺失时，不是在总线上广播，而是发送无效消息和执行写回操作。同监听控制器一样，我们假设把共享Cache块的写操作视为写缺失；实际上，这样的事务处理可以视为所有权请求或更新请求，并能在不要求取回数据块的情况下传递所有权

图 4.21 中的操作基本上与监听协议中的相同：不仅状态是完全相同的，而且激励也几乎是相同的。在监听协议中，写缺失操作规程是通过在总线（或其他网络）上广播实现的，而在目录协议中是通过目录控制器有选择地发送数据获取和无效操作来实现的。和监听协议一样，任何 Cache 块要执行写操作时必须处于独占状态，而且共享数据在存储器中的副本必须更新。

在目录协议中，目录实现了一致性协议的另一半工作。发送到目录的消息引发两种不同类型的操作：一种是更新目录状态，另一种是发送其他消息来响应请求。目录的状态代表了数据块的两个标准状态。与监听方案不同，这些状态针对被缓存的存储器块，而不是针对单一 Cache 块的。

存储器可以不被任何节点缓存，也可以放在多个节点的 Cache 中并使之处于可读的共享状态，或只放在一个节点的 Cache 中并使之处于可写的独占状态。除了每个块的状态外，目录还必须跟踪拥有数据块副本的处理器集合；我们用一个称为共享者的集合来完成这个功能。在具有小于 64 个节点（可代表 2~4 倍的处理器数目）的多处理器系统中，这个集合通常是一个位向量。在较大型的多处理器系统中，就需要其他的技术了。对目录请求的响应不仅需要更新共享者集合，还要读取该集合以执行无效操作。

图 4.22 显示了目录对收到的消息做出应答时所采取的动作。目录会接收到三种不同的请求：读缺失、写缺失和数据写回。目录发送的应答消息用黑体字表示，而共享者集合的更新用黑斜体字表示。因为所有的激励消息都来自外部，所以，所有的动作都用灰体字显示。这个简化的协议假设一些动作是原子的，比如请求数据以及将数据发送到其他节点；具体实现时不能采用这些假设。

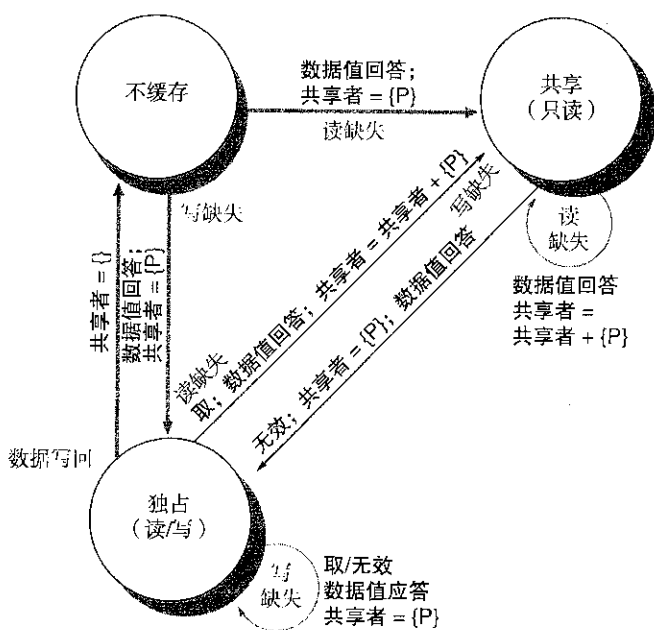


图 4.22 与单个 Cache 状态转换图具有相同状态和结构的目录状态转换图。因为所有的动作都来自外部，所以它们都以灰体字表示。黑体字表示目录响应请求所做的动作

为了理解这些目录操作，首先来研究一下每个状态可以收到的请求和采取的动作。如果一个数据块还未被缓存，那么存储器中的副本就是当前值，所以可能对这个块产生的请求是

- **读缺失**：存储器向发出请求的处理器送回所要求的数据，而发送请求的节点成为唯一的共享节点。块的状态设为共享。

- **写缺失**: 向发出请求的处理器送回数值并使它成为共享节点。数据块设成独占状态, 指明这是唯一有效的 Cache 副本。共享者集合中指明所有者。

当数据块处于共享状态时存储器中的数值也是最新的, 所以会发生两个同样的请求:

- **读缺失**: 存储器向发送请求的处理器送回所要求的数据, 然后将发送请求的处理器放到共享集中。
- **写缺失**: 向发送请求的处理器送回数值。向共享集合中的所有处理器发送无效消息, 并且共享者集合中要保存发送请求的处理器标识。数据块设置成独占状态。

数据块处于独占状态时, 块的当前值保存在由共享者集 (即所有者) 所指明的处理器的 Cache 中, 因此有三个可能的目录请求:

- **读缺失**: 向所有者处理器发送数据消息, 这将使所有者 Cache 中该块的状态转为共享, 并且由所有者向目录发送数据。在目录中该数据被写入存储器并发送回发出请求的处理器上。发出请求的处理器的身分会被添加到共享者集合中, 这时集合中仍然会有所有者处理器的身份 (因为此处理器中含有可读的副本)。
- **数据写回**: 由于此时所有者处理器将数据块改写, 因此必须要执行写回操作。这使得存储器副本得到更新 (实质上主目录成为了所有者), 并且不再缓存数据块, 且共享者集合为空。
- **写缺失**: 数据块有了新的所有者。向旧所有者发送消息使 Cache 将该数据块设置为无效, 并把值发送到目录中, 再通过目录把数据发送到发出请求的处理器上。发出请求的处理器成为新所有者。共享者集合只保留新所有者的标识, 而块仍然处于独占状态。

图 4.22 中的状态转换图与监听协议中一样是简化的。在使用目录及网络而不是总线来实现监听模式的情况下, 协议必须要考虑非原子性的存储事务问题。附录 H 深入地探讨了这些问题。

另外, 实际多处理器系统中使用的目录协议需要优化。特别是在协议中, 发生对独占状态数据块的读缺失或写缺失时, 该块首先被送到主节点的目录。在那里存入主存储器中并且被送到发出请求的节点上, 商用多处理器系统中使用的许多协议把数据从所有者节点直接送到请求节点上 (同时也执行主节点的写回), 这样的优化通常增加了协议的复杂性, 因为这样做既增加了死锁的可能性, 又增加了必须去处理的消息类型。

为了实现目录机制, 需要解决的问题有很多, 绝大多数与 149 页开始时讨论的监听协议中遇到的主要问题相同。另外还有一些新的问题, 我们会在附录 H 中讨论。

4.5 同步: 基本要素

一般来说, 同步机制是通过用户层的软件例程来构造的, 而这些例程要使用硬件提供的同步指令。在规模较小的处理器系统或竞争较少的环境中, 关键是要在硬件上支持一条不可中断的指令或指令序列, 这一指令 (或指令序列) 能以原子操作的方式取回数据并修改其值。软件同步机制是利用上述由硬件提供的这种能力构造的。本节关注锁和解锁同步操作的实现。锁和解锁可以被直接用来实现互斥机制, 以及实现更加复杂的同步机制。

在大规模多处理器系统或竞争较多的环境中, 由于竞争造成额外的时延, 并且这种时延在多处理器系统中存在的可能性更大, 使得同步可能成为一个性能瓶颈。本节讨论基本的同步机制, 并会在附录 H 中扩展到更大量的处理器中。

基本硬件原语

在处理器中实现同步的关键,在于要有一个能够以原子方式对存储器执行读写操作的硬件原语集合,如果没有这种支持,构造基本的同步原语将会付出非常大的代价,并且处理器个数越多,付出的代价也越大。有很多能够代替基本硬件原语的方法。这些方法也都支持原子方式的读写操作并同时能够反馈执行结果。这些硬件原语是构造多种不同的用户层同步操作的基本构件,比如锁和障碍。一般来说,系统结构设计者都不希望用户直接使用这些基本的硬件原语,而是由系统程序员用这些原语来构建一个同步库。这种库的构建过程通常是非常复杂且需要技巧的。下面首先介绍一个这样的硬件原语,然后讲述如何利用它来构造基本的同步操作。

原子互换是一个典型的构建同步原语的操作,它将一个寄存器中的值和一个存储器中的值进行互换。为了理解怎样用它来构造一个基本同步操作,先构造一个简单的锁,其中0表示该锁可以占用,而1表示该锁不能占用。如果处理器想占用这个锁,可以通过将寄存器中的1与该锁在存储器中的值互换来实现。如果已经有其他处理器占用了该锁,返回结果为1,否则为0。如果返回结果为0,锁的数值被设置成1。这样在这个处理器将锁释放之前,其他处理器无法占用这个锁。

例如,考虑两个处理器试图同时进行互换操作的情况:这打破了竞争,因为只有一个处理器能首先执行互换并得到返回值0,而第二个处理器执行互换时将得到返回值1。使用互换或交换原语实现同步的关键问题是操作需要具有原子性:互换是不可割裂开来的,且两个同时进行的互换将由写串行机制进行排序。用这种方式,两个试图设置同步变量的处理器不可能同时完成设置。

还有很多其他原子性的原语也能用来实现同步。它们都有一个关键的属性,就是在执行读写存储器操作时,能向调用者反馈这两个操作是否以原子方式执行。许多旧的多处理器系统使用的一个操作是**测试并置位**,它首先对数值进行检测,如果该值通过了检测则执行设置。例如,可以定义一个测试0和设置1的操作。它的使用方法与原子互换类似。另一个原子性同步原语是“取并增加”,它返回存储器中的值并以原子操作的方式使存储器中的值加1。如果用0表示同步变量未被占用,就可以像使用互换一样使用**取并增加**操作。下面会讲到这类操作还有其他用途。

实现对存储器的单一原子操作存在一些困难,因为它要求对存储器的读和写都在一个独立的、不可中断的指令中完成。这会使一致性的实现变得更加复杂,因为硬件不允许在读和写之间有其他操作,同时又要保证不死锁。

另一个方法是使用一对指令,其中根据第二条指令的反馈可以得知这对指令是否以原子方式执行。如果处理器执行的其他所有操作都在这对指令之前或之后进行,那么这对指令可以看做是原子操作。因此,当这对指令以原子方式执行时,就没有处理器能在两个指令之间改变数值。

这对指令使用一个特殊的加载(称为**链接加载**)和一个特殊的存储(称为**条件存储**)。这两条指令是按顺序使用的:如果由链接加载所指的地址的值在条件存储之前改变了,则条件存储失败。如果处理器在这两个指令之间做了上下文切换,则条件存储也失败。按照定义,条件存储在成功时会返回1,而在失败时返回0。因为链接加载会返回初始值而条件存储仅在成功时返回1,所以下面的指令序列在R1所指向的存储器地址上实现原子互换:

```
try:  MOV    R3,R4      ; 移动需要互换的值
      LL     R2,0(R1)   ; 链接加载
      SC     R3,0(R1)   ; 条件存储
      BEQZ   R3,try     ; 存储转移失败
      MOV    R4,R2      ; 把加载值放入 R4
```

这个序列结束时 R4 中的值和 R1 指定的地址中的值进行了原子互换（忽略一切来自转移延迟的影响）。只要处理器在 LL 和 SC 指令序列之间修改了存储器的值，SC 就在 R3 中返回 0，然后会转到 try 处重新执行。

链接加载和条件存储机制的另一个优点在于它能用于构造其他同步原语。例如，下面是原子方式的取并增加操作：

```
try:  LL      R2,0(R1)  ; 链接加载
      DADDUI  R3,R2,#1  ; 增加
      SC      R3,0(R1)  ; 条件存储
      BEQZ    R3,try     ; 存储转移失败
```

这些指令都是通过用一个寄存器来跟踪 LL 指令中所指定的地址来实现的，这个寄存器通常称为**链接寄存器**。如果发生中断，或者链接寄存器所指向的地址对应的 Cache 数据块变为无效（例如，另一条 SC 语句），就要清除链接寄存器。SC 指令只检查它的地址是否与链接寄存器中的地址匹配；如果匹配，则成功，否则失败。因为条件指令之间加入其他指令时要非常谨慎，只有寄存器-寄存器这样的指令不会产生安全问题；否则就有可能导致死锁，使处理器无法完成 SC。另外，在链接存储和条件加载之间加入的指令要尽可能少，这样可以减少无关事件或处理器争用等引起条件存储频繁失败的可能。

用一致性实现锁

一旦有了原子操作，就可以利用多处理器的一致性机制来实现**自旋锁**（即处理器通过循环来不停尝试获得锁，直到成功为止）。有以下两种情况会用到自旋锁。第一，程序员要求占用锁的时间很短；第二，程序员要求锁在可用时，锁定过程的时延较低。因为自旋锁要阻塞处理器并一直循环等待锁被释放，所以自旋锁在某些环境下是不适用的。

若不存在 Cache 一致性问题，所能使用的最简单的方法是在存储器中保存锁变量。处理器可以不停地使用原子操作尝试占用锁，或者说互换，以及检测锁是否可用。要释放锁，处理器只要将 0 值存储到锁中即可。下面的代码序列使用原子互换来锁定 R1 指定的自旋锁：

```
      DADDUI  R2,R0,#1
lockit: EXCH   R2,0(R1)  ; 原子锁定
      BNEZ    R2,lockit  ; 已经锁定？
```

如果多处理器支持 Cache 一致性，就可以利用一致性协议把锁放入 Cache 来保证其一一致性。这样做有两个优点。第一，它使“自旋”（在一个紧凑的循环中不断检测和尝试占用锁）过程的实现能够在本地 Cache 副本中完成，而不必要每次尝试占有锁时都进行全局存储访问。第二个优点来自对锁的访问的局部性，也就是说最后一个使用该锁的处理器很快会再次使用它。在这种情况下，把锁驻留在使用它的那个处理器的 Cache 中可以大大减少获取锁需要的时间。

为了实现第一个优点，即每次试图获取锁时能够在本地 Cache 副本中完成而不必访问存储器，需要对上面的简单自旋过程做一点修改。上述循环中的每次直接互换的尝试都要执行一次写操作，如果多个处理器试图占有一个锁，则每个处理器都要执行写操作，其中大多会导致写缺失，因为每个处理器都要以独占状态占用该锁变量。

下面修改一下自旋锁过程，修改后它会对锁的本地副本进行循环检查，直到发现可以获取该锁。然后它会通过交换操作占用该锁。处理器会首先读入锁变量以检测其状态。处理器会不停地读入并检测，直到读入的数值表示锁已被解锁为止。然后处理器会同其他也在“自旋”等待的进程竞争，看谁能首先锁定该变量。所有进程都使用交换指令来读入以前的值并把 1 存到锁变量中。唯一

一个获胜者将会读入0, 而失败者则会看到由获胜者存入的1。失败者会继续设置变量为锁定值, 但这已无关紧要。获胜的处理器继续执行后面的代码, 完毕后把0放入锁变量以释放锁, 之后新一轮的竞争就又重新开始了, 下面是实现这个自旋锁的代码 (0表示未锁定而1表示锁定):

```
lockit: LD      R2,0(R1)      ; 锁的加载
        BNEZ    R2,lockit    ; 不能获取
        DADDUI  R2,R0,#1     ; 加载锁定值
        EXCH    R2,0(R1)     ; 交换
        BNEZ    R2,lockit    ; 如果不是0就跳转
```

下面研究一下这个“自旋锁”方案是怎样使用Cache一致性机制的。图4.23列出了多个进程尝试用原子交换来锁定变量时处理器和总线或目录的操作。一旦占用锁的处理器把0放入锁中, 所有其他的Cache副本就变为无效而且必须取回新值来更新其副本。其中只有一个Cache首先取得了未锁定值(0)的副本然后执行了交换。其他处理器的Cache缺失得到响应之后, 它们会发现变量已被锁定, 因此必须重新检测和自旋。

步骤	处理器 P0	处理器 P1	处理器 P2	锁的一致性状态	总线/目录活动
1	占有锁	自旋, 检测锁是否为0	自旋, 检测锁是否为0	共享	无
2	设置锁为0	(接收到无效信号)	(接收到无效信号)	独占 (P0)	P0 发送锁变量写无效
3		缓存缺失	缓存缺失	共享	总线/目录响应 P2; 执行写回 P0
4		(当总线/目录忙时等待)	锁为0	共享	P2 缓存缺失得到响应
5		锁为0	执行交换, 获得缓存缺失	共享	P1 缓存缺失得到响应
6		执行交换, 缓存缺失	完成交换: 返回0, 并设锁为1	独占 (P2)	总线/目录响应 P2 的缓存缺失; 产生无效
7		交换完成并返回1, 设置锁为1	进入临界区	独占 (P1)	总线/目录响应 P1 的缓存缺失; 产生写回
8		自旋, 检测锁是否为0			无

图 4.23 P0, P1 和 P2 三个处理器的 Cache 一致性步骤和总线通信。图中使用写无效一致性协议。开始时 P0 占有锁 (步骤 1), P0 退出并释放锁 (步骤 2)。P1 和 P2 竞争看谁能在交换期间读入未锁定的值 (步骤 3~5); P2 胜出并进入临界区 (步骤 6 和步骤 7), 而 P1 失败所以开始自旋等待 (步骤 6 和步骤 7)。在实际的系统中, 这些事件所需的时间会比 8 个时钟周期长得多, 因为获取总线和缺失应答需要更长的时间

这个例子说明了链接加载/条件存储原语的另一个优点: 写和读操作是显式分离的。链接加载不会引起任何总线通信。这样就有如下的简单代码序列, 它的特征和使用互换的优化版本 (R1 中指定锁的地址, LL 取代了 LD, SC 取代了 EXCH) 相同:

```
lockit: LL      R2,0(R1)      ; 链接加载
        BNEZ    R2,lockit    ; 不能锁定的自旋
        DADDUI  R2,R0,#1     ; 值被锁定
        SC      R2,0(R1)     ; 存储
        BEQZ    R2,lockit    ; 如果存储失败则跳转
```

第一个跳转构成了自旋循环；第二个跳转则解决了两个处理器同时发现锁可用时引起的竞争。

虽然自旋锁方案简单且具有吸引力,但由于释放锁时产生的通信量问题,它难以扩展到多个处理器的情况。附录H中就这个问题以及较大处理器数量的其他问题进行了专门的讨论。

4.6 存储器连贯性模型：介绍

Cache一致性确保了多个处理器所看到的存储器视图是连贯的,但是它并未说明存储器视图必须达到怎样的一种连贯性。这里所说的连贯性是指被一个处理器修改过的值,在何时可以被另一个处理器访问。既然处理器之间是通过共享变量(其中包括用于数据值的共享变量和用于同步操作的共享变量)来进行通信的,那么上述问题就归结为:一个处理器应该以何种次序来查看被别的处理器写过的数据值,才能保证连贯性。由于查看的方式就是读操作,因而上述问题则变为:在不同的处理器对不同的单元进行读操作和写操作时,应加上何种属性才能确保连贯性。

虽然这个问题看起来很简单,但是通过下面这个简单的例子我们可以看到,它实际上是个非常复杂的问题。下面并排的是进程P1和P2两段代码:

```
P1:      A = 0;          P2:      B = 0;
        .....
        A = 1;          .....
        B = 1;
L1:      if (B == 0) ... L2:      if (A == 0) ...
```

假定进程运行于不同的处理器,两个处理器的Cache中都已包含单元A和B,且其初值都为0。如果写操作总是立即生效并能立即被别的处理器看到,那么两个if语句(L1和L2)的条件就可能同时满足,因为执行到if语句就意味着A和B已被赋值1。但是如果写无效被延迟,并且允许处理器在时延期间继续往下执行,那么P1和P2就有可能在尝试(分别地)获取A和B之前没有看到无效信号。问题是:是否允许这种行为?如果允许,应在什么样的条件下进行?

存储器连贯性最直接的模型是顺序连贯性模型。它要求程序每次执行的结果都是一样的,就像每个处理器执行访问存储器的操作是顺序的,不同处理器之间的操作则是随意交替进行的。在上例中顺序连贯性模型没有考虑一些不太明显的执行顺序。因为在顺序连贯性模型中,if语句初始化之前赋值语句已经完成了。

实现顺序连贯性最简单的方法是,让处理器将存储器访问操作的完成时延到由该访问操作引起的所有无效动作完成为止。当然采用时延下一个存储器的访问操作直到前一个操作完成的方法也同样有效。存储器连贯性问题涉及的是不同变量之间的操作,即两个被排序的访问操作实际上访问的是不同的存储器的单位。上述例子中,对A和B的读操作($A == 0$ 或 $B == 0$)必须时延到前一个写操作($B = 1$ 或 $A = 1$)完成之后进行。在顺序连贯性模型中,不可以简单地将写操作放入写缓冲区,然后继续执行操作。

虽然顺序连贯性模型所编写出的程序很简单,但它却降低了潜在的性能,尤其是在有大量处理器的多处理器系统中或是有很长的通信时延时更为明显,下面的例题说明了这一点。

例题 假设一个处理器的写缺失需用50个周期来建立该缺失块的拥有权,完成后每个无效信号的发送需要10个周期,从无效动作完成到收到应答还需用80个周期。假设四个处理器共享一个Cache块,在顺序连贯性模型中,一个写缺失操作要对执行写操作的处理器造成多长时间Y+的延迟呢?假设目录控制器在完成无效操作之前,必须明确地得到无效信号的应答。同时假设处理器为写缺失建立拥有权后,无须等待无效信号就可以继续运行,那么该操作要花费多少时间呢?

解答: 如果要等待无效信号的应答, 则每个写操作花费的时间等于建立拥有权的时间加无效操作的时间。因为无效操作可以重叠进行, 故只需考虑最后一个无效操作所用的时间, 它在建立拥有权的 $10 + 10 + 10 + 10 = 40$ 个周期后开始。因此总的时间为 $50 + 40 + 80 = 170$ 个周期。相比较而言, 建立拥有权只需 50 个周期, 采用适当的写缓冲技术, 处理器甚至可以在拥有权建立之前继续执行。

为了提供更好的性能, 研究人员和设计人员探讨了两种不同的方法。第一种是开发难度很高的实现方法, 它在保证顺序连贯性的同时采用时延隐藏技术来减少开销; 在 4.7 节会讨论这个问题。第二个是开发一些限制较少且允许硬件速度更快的存储器连贯性模型。这些模型会影响到程序员对多处理器的视图。所以在讨论这些模型之前, 首先看看程序员所希望看到的是什么。

程序员的视角

虽然顺序连贯性模型有性能上的不足, 但从程序员的角度来看, 它具有简单性的优点。这其中的挑战在于开发一个解释起来简单而实现起来又具有高性能的程序设计模型相当困难。

为了实现起来更容易, 我们假设在这样的编程模型中程序是同步的。程序同步是指所有对共享变量的访问都由同步操作来排序, 用同步操作对数据引用操作进行排序是指在任何可能的运行过程中, 某处理器对一个变量的写操作和另一个处理器对该变量的访问(读或写)操作都被一对同步操作隔离开来, 其中一个同步操作由写操作的处理器在执行写操作后执行, 另一个同步操作由第二个处理器在执行访问操作之前执行。不使用同步操作对变量的更新进行排序的情况称为数据竞争, 因其运行结果和处理器的相对速度有关。这如同硬件设计中的竞争那样, 其结果是无法预知的。因此同步程序就有了另一个名字: 无数据竞争程序。

这里看一个简单的例子, 两个不同的处理器对一个变量进行读操作和更新操作。每个处理器使用一个加锁(lock)和一个解锁(unlock)把读操作和更新操作包在里面, 以确保更新操作的互斥性和读操作的连贯性。这样, 每个处理器的写操作和另一处理器的读操作都被一对同步操作——解锁(在写操作之后)和加锁(在读操作之前)——隔离开了。当然, 即使两个处理器的写操作中间不插入读操作, 写操作也必须要采用同步操作隔离开。

大多数程序是同步的。这是被广泛认同的。因为如果访问不同步, 则程序的行为就难以确定, 这样一来, 执行的速度将决定哪个处理器赢得数据竞争, 进而影响程序的结果。即使在顺序连贯性模型中, 分析这种程序也是非常困难的。程序员可以通过建立自己的同步机制来尝试确保程序的顺序执行, 但这需要很高的技巧且容易造成错误, 而且在系统结构层次上也可能是不支持的, 也就是说下一代多处理器上可能不能运行这些程序。所以几乎所有的程序员都选择使用同步库, 这些同步库是正确无误的, 且已根据多处理器和同步的类型进行了优化。最重要的是, 标准同步原语的使用确保了即使系统结构采用的是比顺序连贯性模型更宽松的连贯性模型, 同步程序也会表现得如同在硬件上实现了顺序连贯性模型一样。

非严格连贯性模型: 基本概念

非严格连贯性模型的关键在于允许读写操作打乱次序完成, 但要用同步操作来保证排序原则, 使得一个同步程序的表现和处理器使用顺序连贯性模型时的表现一样。根据放宽的读写顺序的内容, 可以将非严格连贯性模型划分为很多不同的种类。使用特定的一组规则来阐述这一排序, 如 $X \rightarrow Y$ 表示操作 X 必须在操作 Y 完成之前结束。顺序连贯性要求支持所有可能的 4 种排序: $R \rightarrow W$, $R \rightarrow R$, $W \rightarrow R$, $W \rightarrow W$ 。对于不同的非严格模型, 放宽的排序也不同;

1. 放宽 $W \rightarrow R$ 排序, 由此产生了一种称为完全存储排序的模型, 或者处理器连贯性模型。因为这种排序原则维持了写的次序, 许多运行于顺序连贯性模型下的程序可以不用额外的同步操作而运行在这种模型下。
2. 放宽 $W \rightarrow W$ 排序, 由此产生一种称为部分存储排序的模型。
3. 放宽 $R \rightarrow W$ 和 $R \rightarrow R$ 排序, 由此产生包括弱排序连贯性模型、Power PC 连贯性模型和释放连贯性模型在内的多种模型, 具体是哪种模型要看排序约束的细节和同步操作保证次序的方式。

通过放宽对这些次序的要求, 处理器可能在性能上获得显著的提升。然而, 描述非严格连贯性模型非常复杂, 其中包括放宽不同排序的优势和复杂性, 还包括对写操作完成的精确定义, 以及处理器何时能看到自己所写的值。要获得更多的关于非严格模型的复杂性、实现问题和潜在性能优势的信息, 强烈推荐读者参阅 Adve 和 Gharachorloo[1996]。

关于连贯性模型的最后小结

目前的大部分多处理器系统都支持某种非严格连贯性, 这些模型包括从处理器连贯性到宽松的连贯性的一系列模型。由于同步机制和多处理器特性密切相关, 且容易造成错误, 因而希望大部分的程序员都使用标准的同步库来构造同步程序, 他们不需要考虑选择非严格连贯性模型就能获得较高的性能。

另一个不同的观点认为非严格连贯性模型的性能优势在很大程度上能通过顺序连贯性或者处理器连贯性获得, 不过这当中有些猜测的成分。在下一节中将讨论这部分内容。

支持非严格连贯性模型的关键在于编译器的角色以及编译器对潜在共享变量的存储器访问的优化。下一节也会讨论这个问题。

4.7 相关问题

由于多处理器重新定义了许多系统特性 (如性能评估、存储器时延以及可扩展性的重要性等), 因此在整个系统结构方向上引入了许多饶有趣味的设计问题, 这些问题对于软硬件的设计都产生了影响。本节将给出几个和存储器连贯性问题相关的例子。

编译器优化和连贯性模型

为存储器连贯性定义一个模型还有另外一个原因, 这就是要对编译器优化共享数据指定一个范围。在显式的并行程序中, 除了同步点被清楚地定义出来且程序是同步的以外, 编译器不能交换对两个不同共享数据项的读操作和写操作的顺序, 因为这种交换可能会改变程序的语义。这就限制了一些甚至是相对简单的优化技术的使用, 例如共享数据的寄存器分配技术, 因为这种处理通常会互换读操作和写操作。在隐式的并行程序中——如用高性能 FORTRAN (HPF) 写的程序——程序必须是同步的, 而且同步点是已知的, 故不存在这些编译优化的问题。

在严格连贯性模型中用推测来实现时延隐藏

如第2章中所提到的, 推测可以用来隐藏存储器时延。预测也可以用来隐藏严格的连贯性模型所引起的时延, 使其具有多数非严格连贯性模型的优点。问题的关键在于处理器用动态调度来对存储器访问进行重排序, 使其可能乱序执行。存储器访问的乱序执行可能会违反顺序连贯性, 影响程序的执行。借助推测执行处理器的时延提交特性, 可以避免这种问题。假设这个连贯性协议建立在

发送无效信号的基础上,如果在一个存储器访问被提交前处理器收到该存储器访问的无效信号,那么处理器会采用推测恢复来取消计算并从无效的存储器地址重新开始。

如果重排一个处理器的存储器请求所产生的执行顺序可能导致一个和在顺序连贯性情况下不同的结果,处理器将会重新执行。这种方法的关键在于处理器只需保证得到的结果跟所有访问按顺序完成得到的结果是一样的。通过判断什么时候结果可能不同,处理器可以做到这点。这种方法很有吸引力,因为这种推测的重新开始很少发生,唯一可能发生的情况是当非同步的访问实际上导致竞争时[Gharachorloo, Gupta 和 Hennessy 1992]。

Hill[1998]建议将顺序连贯性或处理器连贯性和推测执行结合起来,作为连贯性模型的一种选择。他的依据有三点。第一,顺序连贯性或处理器连贯性如果实现得较好,可以获得非严格模型的大部分优点。第二,这种实现的花费仅比推测执行处理器多一点。第三,这种实现允许程序员考虑采用更简单的顺序连贯性或处理器连贯性的编程模型。

MIPS R1000的设计小组在1990年中期就考虑到了这一点,并利用R10000的乱序执行能力来支持这类顺序连贯性的实现。Hill的论据可能会激发其他人采用这个方法。

另外一个未解决的问题是,在共享变量的存储器访问优化上,编译技术可以带来多大的成效。优化技术的现状,以及共享变量通常通过指针或数组下标访问的事实,都限制了这些优化的采用。如果这项技术变得可用且带来巨大的性能优势,编译器的开发者可能会充分利用更加宽松的编程模型。

包含性及其实现

许多多处理器系统使用多级Cache层次结构来减少对全局互连的要求和由Cache缺失引起的时延。如果Cache同时也能提供多层包含性——即每一级Cache层都是离处理器更远一层Cache的子集——那么就可以采用多层结构降低一致性流量和处理器流量间的竞争,它在监听和处理器访问Cache出现竞争Cache的情况下就会发生。虽然近来由更小的一级Cache和不同的块大小构成的多处理器常常选择不加强包含性,但大多数具有多层Cache的多处理器系统都要确保包含性。这一限制也称为子集特性,因为每个Cache都是其下层Cache的子集。

保留多层包含性似乎无关紧要。让我们看一下两层的例子:在一级Cache中的缺失,无论在二级Cache中命中或缺失,都会使缺失块进入一级Cache和二级Cache,同样在二级Cache中碰到的块无效信号将被送到一级Cache,如果一级Cache中有该块,则也应使其无效。

关键是当一级Cache和二级Cache的块大小不一样时会发生什么情况。选择不同尺寸的块是很自然的,因为二级Cache容量要大得多,而且在缺失时所经历的时延也大得多,因而它要求用的块要更大一些。当块大小不一样时,包含性应该如何自动保持呢?此时二级Cache中的一个块代表了一级Cache中的多个块,二级Cache中的一个缺失导致的一个块替换要引起一级Cache多个块的替换。例如二级Cache的块大小是一级Cache的四倍,则二级Cache的缺失将引起四个一级Cache块的替换。让我们来看一个详细的例子。

例题 假设二级Cache的块大小是一级Cache的四倍,试说明因一个地址的缺失而引起的一级Cache和二级Cache的替换是如何违背包含性特性的。

解答: 假设一级Cache和二级Cache是直接映射的,一级Cache的块大小为 b 字节,二级Cache的块为 $4b$ 字节。设一级Cache包含起始地址为 x 和 $x+b$ 的块,且 $x \bmod 4b = 0$,这表示 x 也是二级Cache的块的起始地址。二级Cache中的一个单独块含有一级Cache的 $x, x+b, x+2b$

和 $x+3b$ 块。如果处理器产生一个对 y 块的访问请求, 该块映射到两个 Cache 中含有 x 的块, 并产生了缺失。由于二级 Cache 缺失, 它取出 $4b$ 个字节, 并且替换了含有 $x, x+b, x+2b$ 和 $x+3b$ 的块, 但一级 Cache 只取 b 个字节来替换含有 x 的块。由于一级 Cache 仍含有 $x+b$, 但二级 Cache 不含有 $x+b$, 故包含性被破坏。

为了在 Cache 块大小不同时仍保证包含性, 在低一级完成替换时, 必须检查所有的高层 Cache, 确保在低层中替换了的块在高层中无效。不同程度的组相连方式产生相同的问题。2006 年设计者在包含性的实施上似乎产生了分歧。Baer 和 Wang[1988] 详细描述了包含性的优点和遇到的问题。

4.8 综合: Sun T1 多处理器

T1 是 Sun 公司于 2005 年作为服务器处理器引入市场的多核多处理器。T1 特别引人注意的一点是, 它几乎完全集中于线程级并行的开发而不是指令级并行的开发。事实上, T1 是近五年多来, 唯一一种商用的单发射桌面或服务器多处理器。T1 不关注指令级并行, 而是把其全部注意力放在了线程级并行, 同时使用多核和多线程技术来提高吞吐量。

每个 T1 处理器包含 8 个处理器内核, 每个内核支持 4 个线程。每个处理器内核由一套简单的六段单发射流水线 (在附录 A 中标准五段 RISC 流水线的基础上加上线程交换阶段) 构成。T1 使用细粒度多线程技术, 在每个时钟周期切换到新线程, 而且由于流水线延迟或 Cache 缺失和在调度中用旁路避免 Cache 缺失的原因, 线程需要等待, 即产生空闲状态。处理器只有在所有的 4 个线程都为空闲或停止状态时才是空闲的。装载和转移操作都会引发只有其他线程才能掩盖的 3 个时钟周期的时延。因为 T1 同样也不十分关注浮点运算的性能, 所以每个处理器上的 8 个内核共享一个浮点功能单元集合。

图 4.24 给出了 T1 处理器的结构。各个内核通过交叉交换机访问 4 个二级 Cache 和共享的浮点单元。一级 Cache 通过目录和每个二级 Cache 块联系起来, 从而实现了一致性。目录的操作和我们在 4.4 节讨论的内容相似, 但常常是用来跟踪哪一个一级 Cache 拥有二级 Cache 块的副本。通过将每一个二级 Cache 同特定的存储器组联系起来, 同时增强特性子集, T1 可以将目录放在二级 Cache 而不是存储器中, 这样就减小了目录的开销。因为一级数据 Cache 是写直达的, 只有无效信息才会被要求; 数据总是可以从二级 Cache 中重新得到。

图 4.25 概括了 T1 处理器。

T1 的性能

使用三个面向服务器的基准测试程序来考虑 T1 的性能, 分别是 TPC-C, SPECJBB (SPEC 的 Java 业务基准测试程序) 和 SPECWeb99。由于 SPECWeb99 基准测试程序不能扩展到 8 核处理器上以充分使用 32 线程, 所以只能运行在 T1 的 4 核版本上; 其余的两个基准测试程序都运行在 8 核 (每内核 4 线程, 一共 32 线程) 的 T1 上。

首先分别考虑在单线程模式和多线程模式下多线程机制对运行的存储器系统性能的影响。图 4.26 给出了在 TPC-C 基准下, 对比每个内核执行 1 个线程和 4 个线程的情况, 可以看到缺失率和观察到的缺失时延都有不同程度的增长。这两种增长都归因于存储器系统中竞争的增加。缺失时延的增幅相对较小说明了存储器系统仍然有未用的容量空间。

正如我们在前面一节阐明的那样, 多处理器负载的性能本质上取决于存储器系统及其与应用之间的交互。对于 T1 来说, 二级 Cache 容量和块大小都是关键的参数。图 4.27 给出了分别将二级

Cache 容量降低 2 倍（开始为 3 MB）和将块大小减小为 32 字节时对缺失率的影响。从中可以明显看出 3 MB 的二级 Cache 比 1.5 MB 的二级 Cache 拥有更显著的优势；如果是 6 MB 的二级 Cache 会得到更大的改进。就像我们看到的，相对于 32 字节的块大小，选择 64 字节大小的块也会降低缺失率，但肯定降低不了 2 倍。因此，使用字节大的块的 T1 会在存储器中产生更多的通信。但是这是否能拥有更显著的优势取决于存储器系统的特征。

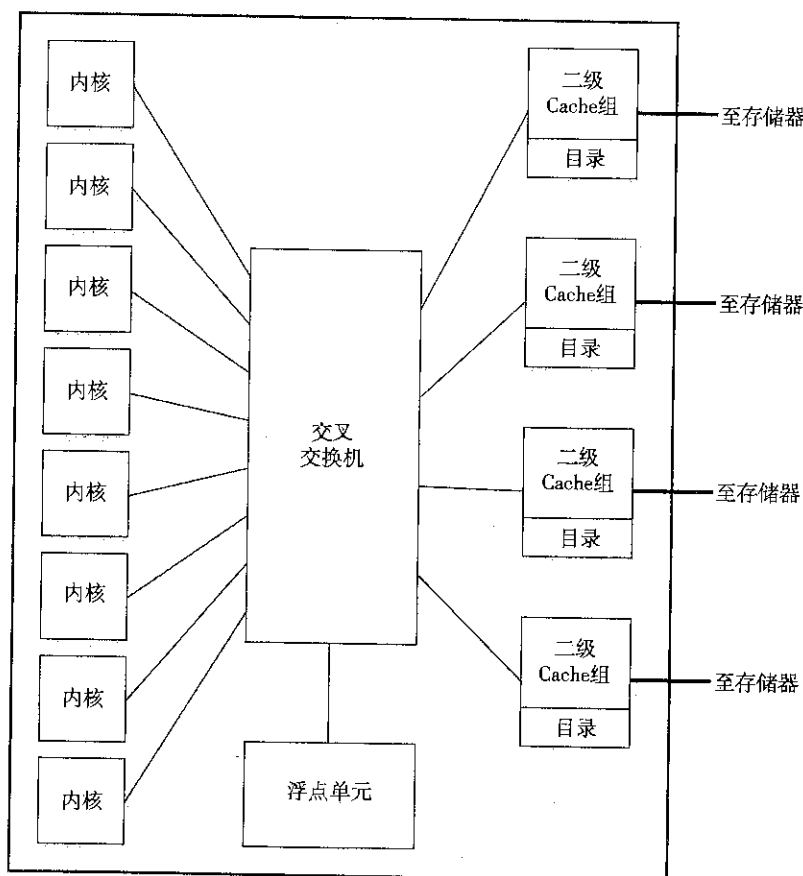


图 4.24 T1 处理器。每个内核支持 4 个线程，并有自己的一级 Cache（16 KB 给指令，8 KB 给数据）。二级 Cache 总共 3 MB，是高效的 12 路组相连方式。Cache 是通过 64 字节 Cache 行交叉存取的

特征	Sun T1
多处理器和多线程支持	每芯片 8 个内核；每内核 4 个线程。高级线程调度。8 个内核共享的浮点运算单元。支持片内多处理器
流水线结构	简单的按序六段流水线，装载和转移操作的延迟为 3 个时钟周期
一级 Cache	16 KB 指令；8 KB 数据。64 字节块大小。假设无竞争的情况下对二级 Cache 每 23 个时钟周期发生一次缺失
二级 Cache	4 个独立的二级 Cache，每个 750 KB 且和存储器组相连。64 字节块大小。假设无竞争的情况下对存储器每 110 个时钟周期发生一次缺失
开始的实现	处理器大小 90 nm；最大时钟频率 1.2 GHz；电源功率 79 W；300 M 个晶体管，面积大小 379 mm ²

图 4.25 T1 处理器的总结

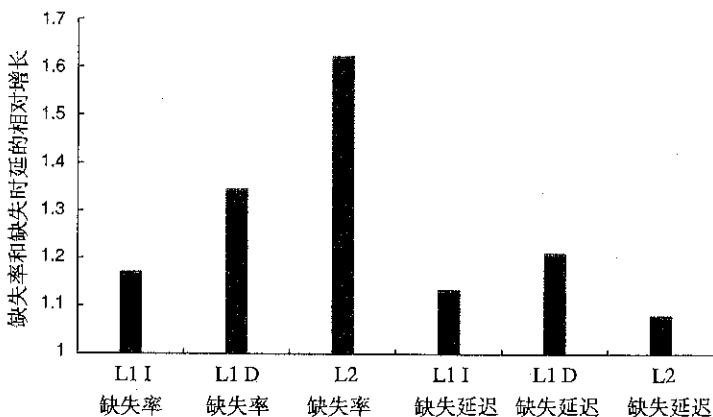


图 4.26 在 TPC-C 基准下，对比每个内核执行 1 个线程和 4 个线程的情况得到的缺失率和缺失延迟相对变化情况。延迟是发生缺失后返回要求数据的实际时间。在 4 线程的情况下，大部分这样的延迟可能会被其他线程的执行所掩盖

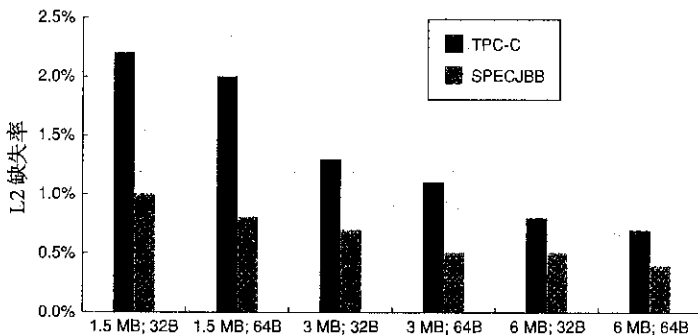


图 4.27 Cache 容量和块大小变化时二级 Cache 缺失率的变化情况。TPC-C 和 SPECJBB 都运行在 8 核（每内核 4 线程）的 T1 上。T1 使用 64 字节行的 3 MB 二级 Cache

就像前面提到的，使用多线程会导致存储器中竞争现象的产生。Cache 容量和块大小是如何影响存储器中的竞争现象的呢？图 4.28 给出了当条件变化与图 4.27 相同时，缺失时延会受到怎样的影响。对 3 MB 或 6 MB 的 Cache 来说，更大的块容量导致更小的二级 Cache 缺失时间。如果缺失率减小了不到 2 倍，这是为什么？原因是现代 DRAM 传送一个数据块比传送单字的时间只多一点点（在下一章中我们可以看到更多的详细情况）；因此，32 字节块的缺失损失只比 64 字节的少一点点。

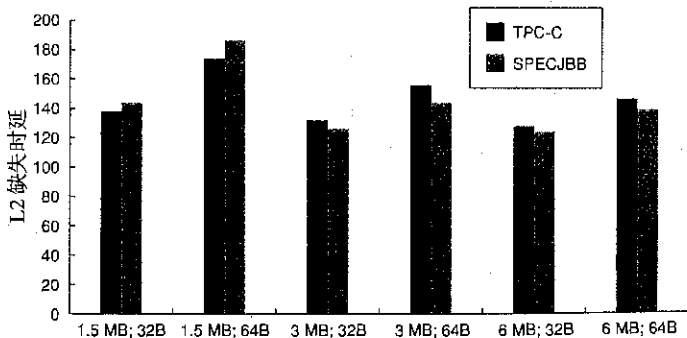


图 4.28 Cache 容量和块大小变化时二级 Cache 缺失延迟的变化情况。尽管 TPC-C 有更高的缺失率，但它的缺失代价只有微小的上升。这是因为 SPECJBB 有更高的脏缺失率，需要十分频繁地写回二级 Cache 行。T1 使用 64 字节行的 3 MB 二级 Cache

整体性能

图4.29显示了每个线程和每个内核的CPI和8处理器芯片的每时钟周期执行指令数(IPC)。因为T1是高级多线程处理器,且每个内核都有4个线程,在充分的并行机制保证下,每个线程理想的有效CPI为4,这意味着每个线程占用四个时钟周期中的一个。每个内核理想的CPI为1。T1的有效IPC为8除以每个内核的CPI。

基准测试程序	每线程 CPI	每内核 CPI	8个内核的有效 CPI	8个内核的有效 IPC
TPC-C	7.2	1.8	0.225	4.4
SPECJBB	5.6	1.40	0.175	5.7
SPECWeb99	6.6	1.65	0.206	4.8

图 4.29 T1 处理器每个线程的CPI、每个内核的CPI、8个内核的有效CPI、8个内核的有效IPC (CPI的倒数)

读者从图中得到的第一印象可能是T1不是特别高效的,因为其有效吞吐量在三个基准测试程序下只有理想状况的56%~71%,但是如果与多发射超标量的性能比较,那么读者可能会改变这种想法。像Itanium 2这样的处理器(更多的晶体管,更高的电源功率,更大的硅面积),如果想要维持每时钟周期4.5~5.7的指令数,以及大大超过两倍于被普遍认可的标准IPC,它需要达到难以置信的指令吞吐量才可以。显而易见,至少对于带线程级并行机制的面向定点运算的服务器应用来说,多核方法比单核多发射处理器方法要更好。下一小节提供了多核处理器之间的一些性能比较。

通过观测每个线程的情况,我们可以理解多线程和并行处理之间的交互。图4.30给出了线程在执行、准备好但未选中、未准备好时所用时钟周期的比例。未准备好并不是说该线程所在的内核已经停顿,内核只有在所有的线程都没有准备好时才停顿。

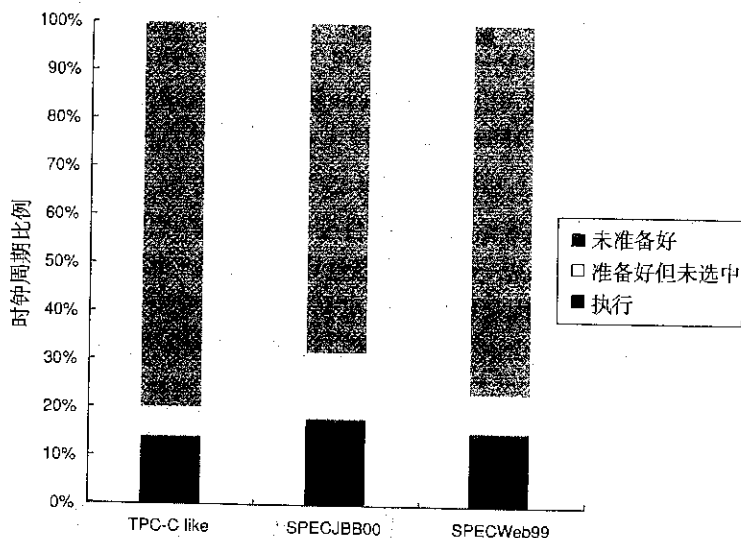


图 4.30 平均的线程状态分析图。执行状态是指线程在该时钟周期发射一条指令;准备好但未选中状态表示线程可以发射,但另外一个线程先被选中;未准备好状态是指线程在等待一个事件的结束(比如流水线延迟或Cache缺失等)

线程未准备好可归因于Cache缺失、流水线时延(源自长时延指令,如转移、装载、浮点数或整数乘除运算指令等)以及其他一些微小的影响因素。图4.31给出了这些原因的相对频率。由于Cache原因造成的线程等待大约占等待时间的50%~75%,包括一级Cache指令缺失、一级Cache数

据缺失以及二级 Cache 缺失,三者比例大致相等。在 SPECJBB 中流水线的可能时延(称为“流水线时延”)是最严重的,这很可能是由于 SPECJBB 中转移指令发生的频率更高。

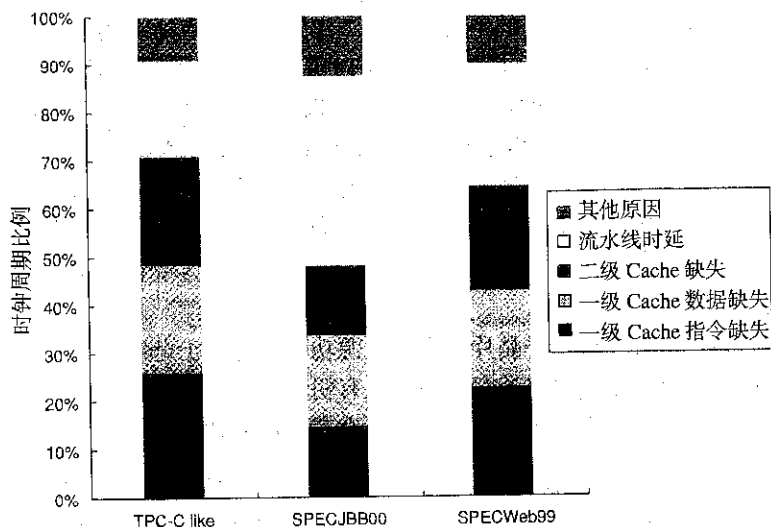


图 4.31 线程未准备好的原因分析图。“其他原因”的组成在各个基准下是不同的:对于 TPC-C 来说,缓冲区存储满是最主要的原因;对于 SPEC-JBB 来说,原子指令是主要的原因;对于 SPECWeb99 来说,上面两个因素都是主要的原因

运行 SPEC 基准测试程序的多核处理器的性能

在最近的多处理器中,T1是唯一一种以线程级并行而不是指令级并行为主要特征的处理器。它使用多线程技术通过简单的 RISC 流水线达到要求的性能,并使用多处理器技术(一个芯片上 8 个内核)来达到服务器应用的高吞吐量。与它不同,双核的 Power 5, Opteron 和 Pentium D 同时使用多发射和多核技术。当然,开发实用的 ILP 需要更强大的处理器,同 T1 相比可能在芯片上需要少一些的内核。图 4.32 总结了这些多核芯片的特征。

特征	SUN T1	AMD Opteron	Intel Pentium D	IBM Power 5
内核	8	2	2	2
每个内核每时钟周期发射的指令	1	3	3	4
多线程	Fine-grained	No	SMT	SMT
Cache	16/8	64/64	12K uops/16	64/32
一级 I/D in KB per core	3 MB shared	1 MB/core	1 MB/core	二级: 1.9 MB shared
二级 per core/shared				三级: 36 MB
三级 (off-chip)				
存储器带宽峰值 (DDR2 DRAMS)	34.4 GB/s	8.6 GB/s	4.3 GB/s	17.2 GB/s
MIPS 峰值	9600	7200	9600	7600
FLOPS	1200	4800(w. SSE)	6400(w. SSE)	7600
时钟频率 (GHz)	1.2	2.4	3.2	1.9
晶体管数量 (百万)	300	233	230	276
晶片面积 (mm ²)	379	199	206	389
电源功率 (W)	79	110	130	125

图 4.32 4 种多核处理器特征的总结

除了侧重点放在 ILP 还是 TLP 上这个不同之处外，这几个处理器和 T1 在设计中还有其他几个本质上的不同之处。其中最重要的是

- 在浮点数支持和操作上有显著的不同。Power 5 主要强调浮点运算的性能，Opteron 和 Pentium D 也分配了大量资源来支持浮点操作，但是 T1 几乎就忽略了这一部分。结果造成了 Sun 公司不可能提供关于浮点应用的基准测试程序结果。用只包括定点运算的基准测试程序对 T1 和包括高性能浮点运算部分（以及相联系的硅原料和电源成本）的其他 3 个处理器做比较是有失公允的。同样，用只包括浮点运算的基准测试程序做比较对 T1 也是不合理的。
- 这些多处理器系统的可扩展性不同，因此对存储器系统设计和外部接口使用的影响也是不同的。Power 5 的设计最具扩展性，Opteron 和 Pentium D 提供了受限制的多处理器支持；T1 则完全不适用于扩展到大系统中。
- 实现技术迥异，这就使得基于晶片大小和电源功率的比较十分困难。
- 对存储器系统以及可用存储器带宽的假设也有明显的不同。对于有高 Cache 缺失率的基准测试程序（TPC-C 及相似程序）来说，有着更大存储器带宽的处理器无疑更有优势。

尽管如此，对以 ILP 为中心的设计和以 TLP 为中心的设计做出各自优劣的评判还是很重要的，所以这里还是试图对这两种方法给出一个性能和效率的量化分析，正如前面分析效率的那种方法一样。图 4.33 给出了使用基准测试程序下 4 种多核处理器的性能数据，这些基准测试程序包括 SPECRate CPU 基准测试程序、SPECJBB2005 Java 业务基准测试程序、SPECWeb05 网页服务器基准测试程序及类 TPC-C 基准测试程序。

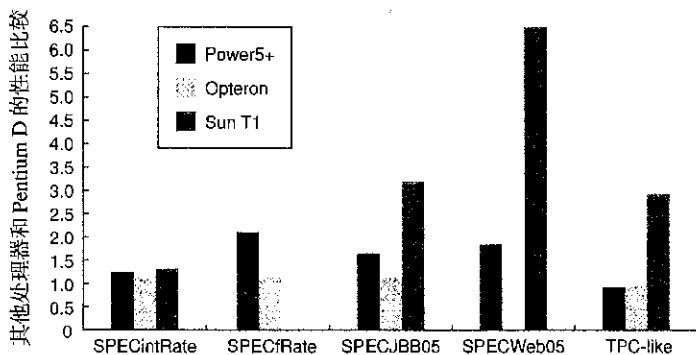


图 4.33 在 SPEC 基准测试程序和类 TPC-C 基准测试程序下，4 个双核处理器的性能显示。所有的数据都以 Pentium D 的数据为标准（其所有数据都为 1）。一些结果是从稍微高一些的配置中（如 4 核双处理器，而不是双核单处理器）得到的，包括 Opteron 的 SPECJBB2005 结果，Power 5 的 SPECWeb05 结果，以及 Power 5，Opteron 和 Pentium D 的 TPC-C 结果。现在 Sun 公司已经拒绝发布 SPECRate 中关于定点或浮点数部分的结果

图 4.34 给出了这 4 种双核处理器基于每个单元晶片面积和每瓦特的性能的效率测量结果，其结果都以 Pentium D 为比较标准。其中最明显的是 Sun T1 处理器在类 TPC-C 和 SPECJBB2005 基准测试程序下性能/瓦特方面的巨大优势。这幅图很清楚地表示出了对于多线程应用来说，TLP 方法要比 ILP 方法在电源功率上更有效率。这是有关 TLP 寻径能在节能环境下提高性能的最力证据。

现在对于加强型 TLP 方法能否在商业上得到成功做出结论还为时过早。如果典型的服务器应用有足够的线程使得 T1 一直处于忙状态，而且每个线程的性能都是可以接受的，那么 T1 这种方法

不会轻易被超越。如果在服务器或桌面环境中单线程的性能仍然很重要, 我们就会看到市场的进一步分化, 将来会出现各种面向完全不同环境的处理器, 包括追求高吞吐量的环境和追求单一线程高性能的环境等。

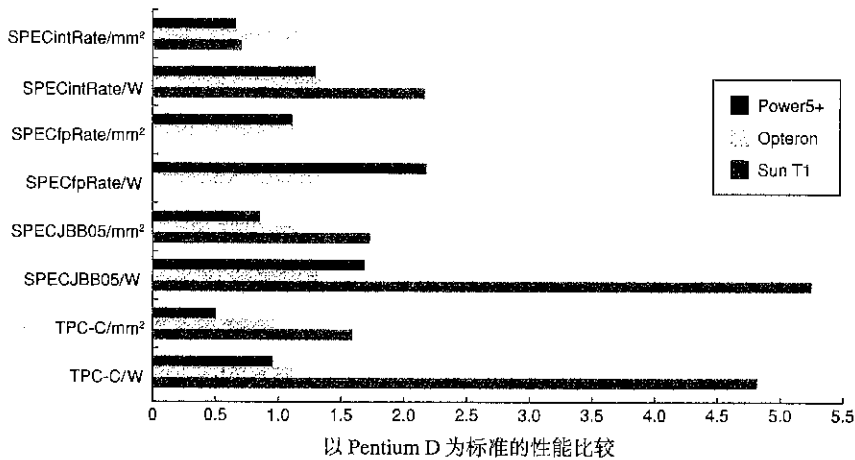


图 4.34 在 SPECRate 下 4 种双核处理器的性能效率，以 Pentium D 为标准度量标准（其为 1）

4.9 谬误和易犯的错误

由于我们对并行计算机的理解还不够透彻, 所以会有很多易犯的错误有待那些严谨的或运气不太好的设计者来发现。由于有关多处理器系统有很多天花乱坠、夸夸其谈的说法, 特别是在高端技术领域, 还有很多普遍的错误看法。我们选了以下这些来讨论:

易犯的错误: 用线性加速比和执行时间的对比来扩展多处理器性能。

“迫击炮发射”图在很长时间里被用来判断并行处理器是否成功, 图中描绘了处理器性能与处理器数目的对比关系, 一般可以看到最初是线性加速, 在到达一个峰值后下落。

虽然加速比能反映并行政程序的某些方面, 但它并不是性能的直接度量标准。首要的问题应该是处理器扩展时的计算能力: 一个程序能够以线性加速比来改善性能, 直到相当于 100 个 Intel 的 486 处理器的性能, 但它却可能会比 Pentium 4 工作站上的串行版本要慢。特别要注意那些浮点计算量大的程序, 那些没有硬件支持的处理器单元可能扩展性会很好, 但是整体性能却很差。

比较执行时间的方法也只有当所采用的算法是各自机器上最好的算法时才算是公平的。在两个机器上比较相同代码的执行看起来是公正的, 但实际上不是。在单处理器上并行政程序可能要比串行版本慢。开发一个并行政程序有时会使算法得到改进, 因此用并行算法和先前流行的最好的串行算法相比较看起来似乎是公平的, 但实际上比较的并不是相同的算法。为了反映这个问题, 有时会引入相对加速比 (同一程序) 和真实加速比 (最好程序) 这些概念。

如果一个程序在 n 个处理器上运行比在相同的单处理器上运行要快不止 n 倍时, 由此可能得出其具有超线性加速比 (superlinear speedup) 性能的结论。虽然确实有过这样“真正的”超线性加速比的例子, 实际上这样的结果可能恰恰表明这样的对比是不公平的。例如, 一些科学应用对于处理器数目的轻微增加 (2 或 4 个到 8 或 16 个) 可以有规则地达到超线性加速比。通常产生这一结果的原因是关键数据结构不适合 2 或 4 个处理器的多处理器的 Cache 总和, 但却适合 8 或 16 个处理器的多处理器的 Cache 总和。

最后,要很好地通过加速比的对比来反映性能是一件非常难的事情,如果处理得不好,具有很强的误导性。对比两个不同多处理器系统的加速比并不会告诉我们相应的多处理器系统的性能。即使在同一多处理器系统上对比两个不同的算法也很难,因为我们要使用真实加速比而不是相对加速比来获得一个正确的对比结果。

谬误: Amdahl 定律不能应用于并行计算机。

在 1987 年,一个研究组织的领导人宣称 MIMD 多处理器打破了 Amdahl 定律(见 1.9 节)。然而这并不意味着这条定律已经被并行计算机推翻;程序中那些被忽视的部分仍然限制了性能。在了解媒体这样报道的根据之前,让我们先来看看 Amdahl[1967]原先是怎么说的:

我们能从这一点得到很明显的结论,那就是为了获得高并行处理速率的更进一步的努力都将是徒劳无功的,除非我们在提高串行处理速率的工作中能有相同数量级的进步。[P.483]

该定律的一个解释是:因为程序总有部分是串行的,因此经济合理的处理器数目会有一个上限——比如说 100 个。所以如果我们能在 1000 个处理器上获得线性的加速比,那么 Amdahl 定律的这种解释就被推翻了。

关于 Amdahl 定律已经被推翻了的说法是建立在规模加速比的应用的基础上的。研究者把测试程序扩展到比原先的数据集大 1000 倍的规模,然后比较扩展后的测试程序在单处理器和多处理器上并行计算的执行时间。因为这是个特殊的算法,程序的串行部分是固定的并独立于输入问题的规模,而程序的其余部分统统都是并行的——因此,在 1000 个处理器上能获得线性的加速比。因为运行时间的增长速度比线性的更快,所以扩展之后程序实际上会运行更长的时间,即使使用 1000 个处理器也不例外。

基于输入可扩展假设得到的加速比和真实的加速比是不一样的,而它似乎具有误导性。因为并行基准测试程序常常运行在不同规模的多处理器上,有必要清楚地说明允许什么样的应用以及如何扩展它们。但是,简单地通过处理器数目来扩展数据大小是很不合适的,对较大规模的多处理器假定问题大小是固定的,同样也是不合适的,因为用户如果有较大规模的处理器,往往就会运行更大和更详细的应用程序。在附录 H 中,讨论了对于大规模多处理器来说扩展应用的不同方法,其中介绍了名为时间限制的扩展模型,这种模型可以扩展应用数据的大小,以便于在一系列的处理器数目中保持执行时间不变。

谬误:使多处理器有较高性价比需要线性加速比。

并行计算被广泛认同的一个主要优点就是比最快的单处理器提供了“更短的时间来计算”。然而许多人坚持认为除非并行处理器能获得良好的线性加速比,否则它们不像单处理器那样划算。这个观点认为,由于多处理器的成本是处理器数目的线性函数,所以小于线性加速比就意味着性能价格比降低,使得并行处理器不如使用单处理器那样划算。

这个观点的问题在于成本不仅是处理器数目的函数,还依赖于存储器、I/O 和系统(机箱、电源、互连等)开销。

Wood 和 Hill[1995]研究了在系统成本中包含存储器的结果。我们使用一个基于最近数据的例子来说明,这些数据是使用 TPC-C 和 SPECRate 基准测试程序得到的。需要说明的是,也可以用并行的科学计算来证明,而且这样的说明可能更有说服力。

图 4.35 给出了在一台 IBM eserver p5 多处理器(配置有 4~64 个处理器)上 TPC-C, SPECintRate 和 SPECfpRate 的加速比。从图中可以看出只有 TPC-C 可以达到比线性更好的加速比。对于 SPECintRate 和 SPECfpRate 来说,其加速比达不到线性加速比,但是成本也做不到,这点和 TPC-C 不同,它们所要求的存储器和磁盘的扩展速度都不到线性的增长速度。

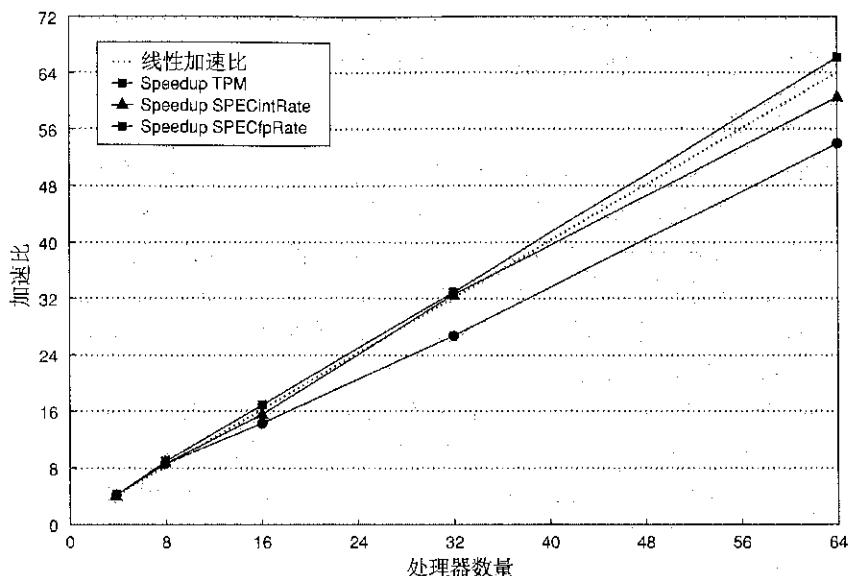


图 4.35 IBM eserver p5 多处理器上三个基准测试程序的加速比，配置分别为 4, 8, 16, 32 和 64 个处理器。虚线表示的是线性加速比

就像图 4.36 所示，更多的处理器数目实际上比 4 处理器配置更划算。未来，随着多处理器成本的下降（与支撑组件的成本相比较而言，包括机箱、电源和风扇等），较大处理器配置的性价比会进一步提高。

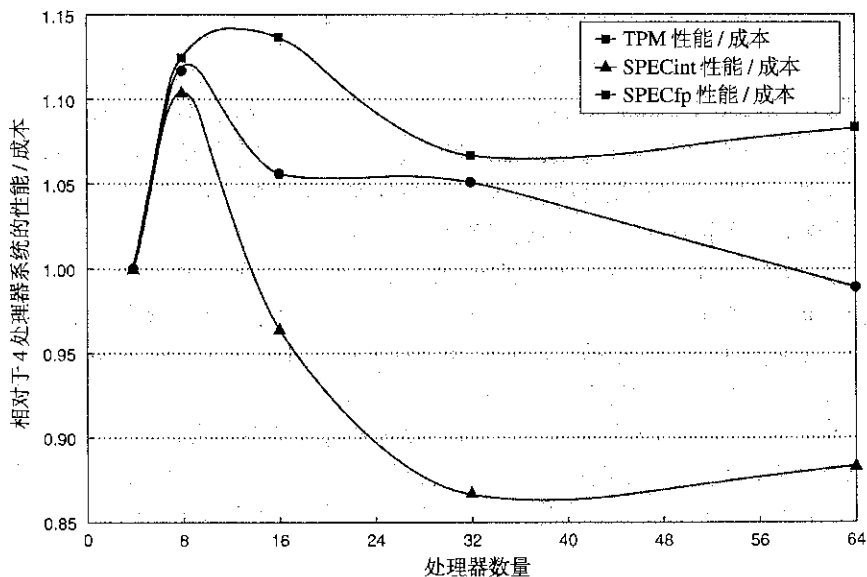


图 4.36 IBM eserver p5 多处理器上运行三个基准测试程序的性价比（配置从 4~64 个处理器），和 4 处理器系统比较后可知，更大的处理器数目和 4 处理器的配置一样划算。对于 TPC-C 来说，配置是在办公环境中使用的，即磁盘和存储器扩展和处理器数目相比较而言接近线性，并且 64 处理器的机器价格差不多是 32 处理器的两倍。相反，磁盘和存储器的扩展速度要慢很多（虽然仍然比达到 64 处理器上最好的 SPECRate 的必需速度要快很多）。特别是，磁盘配置从 4 处理器的一个驱动器变为了 64 处理器的 4 个驱动器（140 GB）。存储器从 4 处理器系统的 8 GB 扩展到 64 处理器系统的 20 GB

在比较两个计算机的性价比时,必须保证包括对总系统成本和所能获得性能的准确评估。对许多有很大存储器要求的应用,这种比较能极大地增加使用多处理器的吸引力。

谬误:可扩展性几乎是免费的。

从20世纪80年代中期开始一直到20世纪90年代末,并行计算的可扩展性已经成为许多研究的焦点和高端处理器开发的重要部分。在这个时期的前半段,人们普遍认为可以把可扩展性加入到多处理器系统中,然后在任何时候,当要从少量处理器向更多处理器扩展时,只要向机器提供处理器就可以了,且不需要其他花费。这个观点实现的困难之处在于开发能扩展到有更多处理器的机器必须要有大量的投入(包括资金和设计时间)来开发处理器间的通信网络,以及满足诸如操作系统支持、可靠性和可配置性等方面的要求。

举一个例子,我们来看一下Cray T3E,它采用3D torus(三维环绕的格形网)作为互连网络,最多可扩展到2048个处理器。在128个处理器时,其峰值二分带宽为38.4 GB/s,平均每个处理器为300 MB/s。但是看一下较小的配置,Compaq的AlphaServer ES40最多可采用4个处理器,互连带宽为5.6 GB/s,每个处理器的带宽为Cray T3E的4倍。此外,Cray T3E中每个处理器的开销要比ES40中的高好几倍。

可扩展性在软件开发中也不是没有成本的:开发可扩展的应用软件,要把更多的注意力放在负载均衡、局部性、对共享资源的潜在竞争和程序的串行(或部分并行)部分。要实现真正意义上的软件可扩展性而不仅仅是针对小的内核或者是像玩具那样的小程序,且能克服处理器数目大于5时而引起的问题,这些都是具有挑战性的课题。将来更好的编译器技术和性能分析工具或许能有助于这个关键问题的研究,但过去的30年中在这个问题上的进展不大。

易犯的错误:软件开发中不用考虑利用或优化多处理器结构。

软件开发落后于大规模并行机器已经有很长的历史了,这或许是由于解决软件问题要困难得多。我们给出了一个例子来说明这个问题的微妙之处,事实上这样的例子不胜枚举。

当为单处理器设计的软件用于多处理器环境时会产生一个需要频繁面对的问题。例如,SGI操作系统用一个单锁来保护页表数据结构,并假设页的分配并不频繁。在单处理器中这不会成为性能上的问题。而在多处理器情况下,它可能会成为某些程序的主要瓶颈。假设有一个包括很多页的程序,它们在程序开始时做初始化(UNIX在静态分配页时就是这样的)。假设程序是并行的,因此有多个进程要分配页。因为页分配要使用页表数据结构,而页表在使用时需要加锁,所以如果所有进程都试图同时分配各自的页(这正是在初始化时所要做的),那么操作系统内核将会被串行化,即使该内核允许在操作系统中同时运行多个线程也无法避免。

这种页表串行化消除了初始化的并行性,并对总体并行性能有巨大影响。这个性能瓶颈即使在多道程序下仍然存在。例如,假如把并行程序分割到各个进程上并运行它们,每个处理器上跑一个进程,这样在各个进程间没有共享(这正是一个用户做的,因为他有理由相信这些性能问题来自他的应用中无意的共享或某些干扰)。很遗憾,页表的锁仍然串行化了所有的进程——即使用多道程序,性能也很差。这个缺陷说明了软件运行在多处理器系统中时会再现这类虽然细微但是严重影响性能的错误。和许多其他关键软件组件一样,操作系统的算法和数据结构必须在多处理器环境中重新考虑。在页表更小的部分上加锁就会有效解决刚才提到的问题。存储器结构中也存在相似的问题,即当实际没有共享发生时会增加一致性通信量。

4.10 结论

在过去超过30年的时间里,研究人员和设计者一直预测单处理器将终结,而多处理器统治的时代将要来临。与此同时,多处理器的增长及其性能的迅速发展很大程度上将多处理器限制在了特定的细分市场中。在2006年,我们已经能够很清楚地看到多处理器和线程级并行将在整个计算领域内起到重要作用。以下所列的几种因素加速了这一变化的发生:

1. 并行处理在某些领域的应用已经得到认可。这些领域里最主要而且最重要的就是科学计算及工程计算。这类领域对于计算能力的渴求几乎是永无止境的。其中某些应用的计算具有天然的并行性。但是,事情也并不是那么简单:即使是针对这些领域,并行处理系统的编程仍然存在很多挑战,我们在附录H中会进一步讨论。
2. 关于事务处理和Web服务的服务器应用和Web服务以及多道程序环境的增长速度很快,通过对独立线程的处理,这些应用可以获得固有的和易于实行的并行机制。
3. 性能经历了20年的快速提升,目前开发ILP的利润正在逐步减少,至少已经能看到这一变化。电源问题、复杂性以及越来越明显的低效问题都迫使设计者考虑它的替代方法。因此开发线程级并行就自然而然地成了未来的选择。
4. 同样,在过去的50年,时钟频率上的改进是来自于改进的晶体管速度。由于技术上的限制和功耗的原因,这种改进正变得越来越困难。因此,开发多处理器并行越来越受到关注。

在本书的1995版中,我们总结这一章时,讨论了当时争论激烈的两个问题:

1. 超大规模的、基于微处理器的多处理器系统会使用什么样的系统结构?
2. 在未来的微处理器系统结构中,多处理器将处于一个什么样的地位?

经过十几年的发展,我们已经可以在很大程度上解决这个问题了。

因为超大规模多处理器没有变成一个主流的或增长的市场,目前唯一经济的办法是使用集群来构建超大规模多处理器。集群的每个节点是单处理器或中等规模的共享存储器多处理器(这种节点只是简单地并入到设计中)。我们在附录E和H中将会讨论集群的设计和节点之间的互连网络。

对第二个问题的回答直到最近才变得清晰起来,但是已经相当明确了。微处理器的性能增长在未来至少5年的发展时间内,将肯定来自于使用多核处理器技术的线程级并行的发展,而不是ILP的开发。事实上,甚至可以看到设计者在未来的处理器中将会选择开发更少的ILP,转而代之的是将关注点放在拥有更多线程级并行的硬件资源上。Sun T1就是在这个方向上迈出的第一步,而且在2006年3月,Intel公司宣布它的下一步多核处理器研究计划将基于更少ILP开发(比Pentium 4 Netburst内核要少很多)的内核。ILP和TLP之间的平衡将可能取决于一系列不同的因素,包括混合应用等。

在20世纪80年代和90年代,随着ILP技术的诞生和发展,能够开发ILP是优化编译器软件成功的关键。类似地,线程级并行能否开发成功也依赖于配套软件系统的开发,就像依赖于计算机系统结构的贡献一样。在过去的三十几年并行软件的发展十分缓慢,所以广泛地开发线程级并行在未来的几年仍然要面临很多挑战。

4.11 历史回顾和参考文献

随书光盘上的K.5节回顾了多处理器和并行处理的发展历史。该节以时间段和系统结构划分,讨论了早期的实验性多处理器和并行处理中的一些大数据库。最近的研究发展也涵盖其中,另外还有参考文献和推荐的进一步阅读材料。

发生变化的块,例如, $P0.B0:(I, 120, 00\ 01)$ 表示处理器 $P0$ 的 $B0$ 块的最后状态为 I , 标示为 120 , 数据字为 00 和 01 。

- a. $[10]<4.2>P0$: read 120
- b. $[10]<4.2>P0$: write $120 \leftarrow 80$
- c. $[10]<4.2>P15$: write $120 \leftarrow 80$
- d. $[10]<4.2>P1$: read 110
- e. $[10]<4.2>P0$: write $108 \leftarrow 48$
- f. $[10]<4.2>P0$: write $130 \leftarrow 78$
- g. $[10]<4.2>P15$: write $130 \leftarrow 78$

- 4.2 $[20/20/20/20]<4.3>$ 监听 Cache 一致性多处理器的性能取决于很多详细的设计问题, 这些问题决定了一个 Cache 怎样才能快速地响应独占或 M 状态块的数据。在一些实现中, 一个 CPU 读取在另一个 CPU Cache 中的独占块时, 会产生读缺失, 这种缺失比读取存储器块时的缺失要快一些。这是因为 Cache 比存储器要更小更快。相反, 在另一些实现中, 存储器块的读缺失要比 Cache 的更快, 这是因为 Cache 的优化一般是为了“前端”或 CPU 调用, 而不是为了“后端”或监听访问。

对于如图 4.37 所示的多处理器来说, 请考虑在一个处理器上执行一系列操作, 这个处理器的状态如下:

- CPU 的读和写命中不产生任何时钟周期的停顿。
- 如果分别满足存储器和 Cache 的条件, CPU 的读和写缺失产生 N_{memory} 和 N_{cache} 时钟周期的停顿。
- CPU 的写命中引发的无效操作 $N_{\text{invalidate}}$ 个时钟周期的停顿。
- 由于竞争或另一个处理器对独占块的访问而导致的块写回, 会导致附加的 $N_{\text{writeback}}$ 时钟周期。

考虑如图 4.38 所示的两种具有不同性能特征的实现方法。

参数	实现 1	实现 2
N_{memory}	100	100
N_{cache}	70	130
$N_{\text{invalidate}}$	15	15
$N_{\text{writeback}}$	10	10

图 4.38 监听一致性时延

下面的每条操作都单独地应用图 4.37 给出的开始状态。为了简化, 假设第二条操作总是在第一条完全结束后开始 (即使是在不同的处理器中):

$P1$: read 110
 $P15$: read 110

对于实现 1, 因为读操作满足 $P0$ 的 Cache, 所以第一个读操作产生 80 个时钟周期的停顿。当 $P1$ 等待块时会产生 70 个时钟周期的停顿, 为了响应 $P1$ 的请求, $P0$ 需要写回存储器块, 这会产生 10 个时钟周期的停顿。然后 $P15$ 的第二个操作会产生 100 个时钟周期的停顿, 这是因为存储器产生了读缺失。所以这一套操作总共产生 180 个时钟周期的停顿。执行下面的每一组操作, 各会产生总共多少个延迟时钟周期?

- a. [20]<4.3> P0: read 120
P0: read 128
P0: read 130
- b. [20]<4.3> P0: read 100
P0: write 108 <-- 40
P0: write 130 <-- 78
- c. [20]<4.3> P1: read 120
P1: read 128
P1: read 130
- d. [20]<4.3> P1: read 100
P1: write 108 <-- 40
P1: write 130 <-- 78

- 4.3 [20]<4.2>许多监听一致性协议都会借助附加的状态、状态转换或总线事务等机制来减少维持Cache一致性的开销。在习题4.2的实现1中,如果这些机制由Cache而不是由存储器提供,不命中产生的时钟周期停顿要少一些。一些一致性协议采用尽力提高这种情况发生频率的办法来改善性能。

一种常见的协议优化方式是引入所有者状态(通常标识为“O”)。当节点只能读所有者状态的块时,所有者状态和共享状态类似;而当其他节点访问所有者状态的块产生读和写缺失时,节点必须提供数据,则所有者状态和修改状态相似。由所有者状态或修改状态的块产生的读缺失要向请求节点提供数据并产生向所有者状态的转换操作。由所有者状态或修改状态的块产生的写缺失要向请求节点提供数据并产生向无效状态的转换操作。当节点取代所有者状态或修改状态的块时,这种优化的MOSI协议只更新存储器。

请画出新的协议状态图,表示新的附加状态和状态转换。

- 4.4 [20/20/20/20]<4.2>对于下面的代码序列和如图4.38所示两种实现的时间参数,分别计算基本MSI协议下的和优化MOSI协议总下的时钟周期停顿数。假设状态转换不要求总线事务,也不导致额外的时钟周期停顿。

- a. [20]<4.2> P1: read 110
P15: read 110
P0: read 110
- b. [20]<4.2> P1: read 120
P15: read 120
P0: read 120
- c. [20]<4.2> P0: write 120 <-- 80
P15: read 120
P0: read 120
- d. [20]<4.2> P0: write 108 <-- 88
P15: read 108
P0: write 108 <-- 98

- 4.5 [20]<4.2>一些应用会先读取大量的数据,然后修改其中的绝大部分或全部数据。基本MSI协议首先找到所有处于共享态的Cache块,然后强制执行一个无效操作,将它们的状态都更新为修改态。这种附加的时延对一些负载来说有明显的影

一种附加的协议优化方法将消除由于单处理器先读后写造成的更新块的需要。这种优化加入了独占状态 (“E”), 表示没有其他节点拥有该块的“副本”, 但是该块尚未被修改。当存储器发生读缺失并且其他节点都没有有效副本时, Cache块就进入独占状态。处理器对该块的读和写操作不会导致进一步的总线通信, 但写操作会将一致性状态转变成修改状态。独占状态和修改状态不同的是节点可以“安静”地替换独占状态的块 (但对于修改状态的块来说必须写回存储器)。同样, 独占状态块的写缺失会将一致性状态转变成共享状态, 但是不再要求节点响应数据 (因为存储器有即时更新的副本)。

请在基本 MSI 协议上加上独占状态及其状态转换, 画出新的 MESI 协议状态图。

- 4.6 [20/20/20/20/20]<4.2>假设 Cache 内容如图 4.37 所示, 并且有图 4.38 所列的实现 1 的各个时间参数。对于下面的代码序列, 分别计算基本 MSI 协议下的和优化 MESI 协议下总的时钟周期停顿数。假设状态转换不要求总线事务, 也不导致额外的延迟时钟周期停顿。

```
a. [20]<4.2> P0: read 100
                P0: write 100 <-- 40
b. [20]<4.2> P0: read 120
                P0: write 120 <-- 60
c. [20]<4.2> P0: read 100
                P0: read 120
d. [20]<4.2> P0: read 100
                P1: write 100 <-- 60
e. [20]<4.2> P0: read 100
                P0: write 100 <-- 60
                P1: write 100 <-- 40
```

- 4.7 [20/20/20/20]<4.5>测试和设置自旋锁是在大多数经济的共享机器中可行的最简单的同步机制。这个自旋锁依赖交换原语来执行装载旧值和存储新值这种原子操作。自旋锁程序重复执行交换操作直到解锁 (如返回值为 0)。

```
tas:      DADDUI    R2,R0,#1
lockit:   EXCH      R2,0(R1)
          BNEZ      R2,lockit
```

自旋锁的解锁只需简单地存储 0 值。

```
unlock:   SW        R0,0(R1)
```

就像在 4.7 节讨论的, 更加优化的测试-测试-设置自旋锁使用 load 命令来检查自旋锁, 允许使用共享变量在 Cache 中自旋。

```
tatas:    LD        R2,0(R1)
          BNEZ      R2,tatas
          DADDUI    R2,R0,#1
          EXCH      R2,0(R1)
          BNEZ      R2,tatas
```

假设处理器 P0, P1 和 P15 都想要获得位置 0x100 处 (如保存在 R1 中) 的自旋锁。假设 Cache 内容如图 4.37 所示, 并且有图 4.38 所列的实现 1 的各个时间参数。为了简化, 假设关键部分有 1000 个时钟周期的长度。

- a. [20]<4.5>使用测试-设置自旋锁, 计算在获得锁之前, 每个处理器大约产生多少个存储器周期停顿?
- b. [20]<4.5>使用测试-测试-设置自旋锁, 计算在获得锁之前, 每个处理器大约产生多少个存储器周期停顿?
- c. [20]<4.5>使用测试-设置自旋锁, 计算大约会发生多少次总线事务?
- d. [20]<4.5>使用测试-测试和设置自旋锁, 计算大约会发生多少次总线事务?

范例分析 2: 交换网络的监听协议

通过这个范例阐明以下概念:

- 监听一致性协议的实现。
- 一致性协议的性能。
- 一致性协议的优化。
- 存储器一致性模型。

范例分析 1 中的监听一致性协议描述了抽象层面上的一致性, 但是其中掩盖了很多重要细节, 并且隐含着这样一条前提假设, 即对共享总线的访问是原子的。因为只有这样才能有正确的操作。一条或多条流水线以及交换互连网络的使用, 极大地提高了高性能监听系统的带宽, 但临时状态以及非原子转换的引入也带来了相当的复杂性。本节的范例分析研究一种高性能监听系统——Sun E6800 的一个松散模型, 其中多个处理器和存储器节点由单独的交换地址和数据网络连接。

图 4.39 给出了这个模型的组织结构图 (中间), 左边是一个处理器节点的详细结构图, 右边是一个存储器模块的详细结构图。和大多数高性能共享存储器系统一样, 这个系统提供多个存储器模块以提高存储器带宽。处理器节点具体包括一个 CPU、Cache 和用来实现一致性协议的 Cache 控制器。CPU 通过请求总线向 Cache 控制器发出读和写请求, 并通过数据总线发送/接收数据。Cache 控制器在本地处理这些请求 (如 Cache 命中时), 缺失时通过 ADDR_OUT 序列将一致性请求 (如 GetShared 请求只读副本, GetModified 请求独占副本) 发送到地址网络中。地址网络使用广播树机制来确保所有的节点都可以看到按全局序列排列的所有一致性请求。所有的节点, 包括请求节点, 都会按与 ADDR_IN 序列相同的顺序收到这一请求 (时钟周期不一定相同)。全局序列是很重要的, 它用来保证所有的 Cache 控制器都以正确的方式操作, 以维持一致性。

该协议保证最多有一个节点回应应该一致性请求, 并将数据消息发送到单独的、无序的点对点数据网络中。

图 4.40 以表格的形式给出了这个系统的一个 (简化的) 一致性协议。由于存在多种状态, 使得状态图很难描述清楚, 所以使用表格来展示一致性协议。每一行对应块的一种一致性状态, 每一列表示影响块的一个事件 (如消息到达或处理器操作), 且每个表格条目表示动作和下一状态 (如果有的话)。一共有两种一致性状态。稳定状态存储在 Cache 中, 即常见的修改 (M)、共享 (S) 或无效 (I) 状态。稳定状态之间的非原子转换会导致临时状态的产生。非原子性的一个重要来源就是因为流水地址网络内部和地址、数据网络之间的竞争。例如, 两个 Cache 控制器可能会在同一时钟周期内向同一个块发出请求消息, 产生竞争后的几个时钟周期内又不知道如何打破这种僵局 (这可以通过监视 ADDR_IN 序列做到, 通过它可以看到请求到来的顺序)。Cache 控制器使用临时状态来记住过去当等待其他操作

时发生了什么。临时状态存储在诸如MSHR的辅助结构中而不是Cache中。在这种协议中，临时状态的名字由开始状态、发展状态以及表明哪种消息仍然未完成的上标编码组成。例如，IS^A状态表示的是块开始在状态I，想要变成状态S，但是在转换之前需要看到它自己的请求消息（如GetShared）已到达ADDR_IN序列。

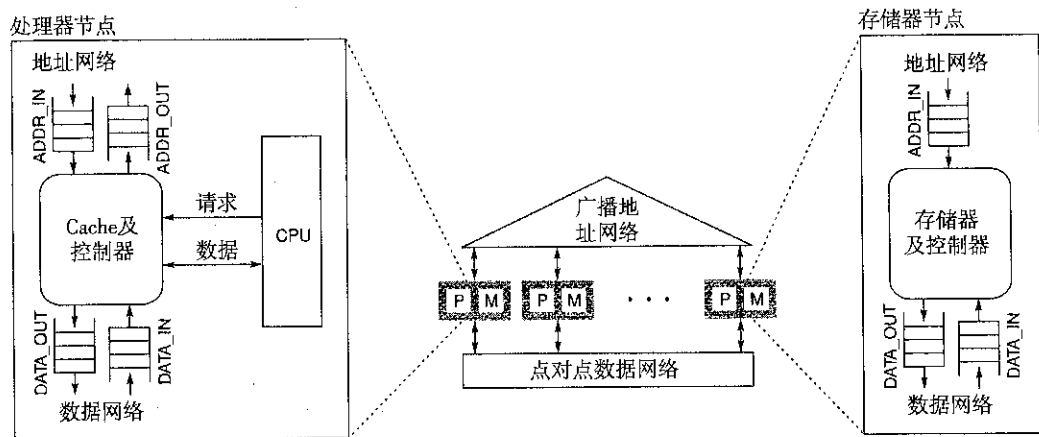


图 4.39 交换互连的监听系统

状态	读	写	替换	OwnReq	其他 GetS	其他GetM	其他 Inv	其他 PutM	数据
I	发送 GetS/ IS ^{AD}	发送 GetM/ IM ^{AD}	错误	错误	—	—	—	—	错误
S	执行读 操作	发送 Inv/ SM ^A	I	错误	—	I	I	—	错误
M	执行读 操作	执行 写操作	发送 PutM/ MI ^A	错误	发送数据 /S	发送数据 /I	—	—	错误
IS ^{AD}	z	z	z	IS ^D	—	—	—	—	保存数据 /IS ^A
IM ^{AD}	z	z	z	IM ^D	—	—	—	—	保存数据 /IM ^A
IS ^A	z	z	z	执行读操作 /S	—	—	—	—	错误
IM ^A	z	z	z	执行写操作 /M	—	—	—	—	错误
SM ^A	z	z	z	M	—	II ^A	II ^A	—	错误
MI ^A	z	z	z	发送数据 /I	发送数据 /II ^A	发送数据 /II ^A	—	—	错误
II ^A	z	z	z	I	—	—	—	—	错误
IS ^D	z	z	z	错误	—	z	z	—	保存数据, 执 行读操作 /S
IM ^D	z	z	z	错误	z	—	—	—	保存数据, 执 行写操作 /M

图 4.40 广播监听 Cache 控制器的转换图

Cache 控制器事件取决于 CPU 请求和收到的请求消息与数据消息。OwnRep 事件表示 CPU 自己的请求已经到达 ADDR_IN 序列。Replacement 事件是一种假的 CPU 事件，当 CPU 读或写触发 Cache 替换时会产生这种事件。图 4.40 详细列出了各个 Cache 控制器事件，每一条目包含一个<动作/下个状态>项。当一个块的状态符合某一行且下一事件满足某一系列时，

P0: replace 110

●无竞争时,数据消息的网络延迟为 L_{data_msg} ,开销为 O_{data_msg} 。

对于下面的代码序列和如图 4.37 所示的 Cache 内容, 以及如图 4.38 所示的实现参数, 计算对每个存储器请求来说, 每个处理器需要产生多少个时钟周期停顿。与之类似, 不同的控制器要占据多少个时钟周期? 为了简便起见, 假设(1) 每个处理器同一时间只有一个存储器操作; (2) 两个节点在相同的时钟周期内发出请求, 如果一个被记录为“赢”, 另一个节点必须因为请求消息开销而停顿; (3) 所有的请求都映射到同一存储器控制器。

- a. [20]<4.2, 4.3> P0: read 120
- b. [20]<4.2, 4.3> P0: write 120 <-- 80
- c. [20]<4.2, 4.3> P15: write 120 <-- 80
- d. [20]<4.2, 4.3> P1: read 110
- e. [20]<4.2, 4.3> P0: read 120
P15: read 128
- f. [20]<4.2, 4.3> P0: read 100
P1: write 110 <-- 78
- g. [20]<4.2, 4.3> P0: write 100 <-- 28
P1: write 100 <-- 48

操作	实现 1	
	时延	开销
send_req	4	1
send_data	20	4
rcv_data	15	4
read_memory	100	20
write_memory	100	20
req_msg	8	1
data_msg	30	5

图 4.41 交换监听一致性时延和开销

4.11 [25/25]<4.2, 4.4>图 4.40 所示的交换一致性协议假设存储器“知道”一个处理器节点是否处于修改状态，因此会用数据响应。实际系统按以下两种方法中的一种来实现：第一种是使用共享的“所有”信号，如果一个其他 GetS 或其他 GetM 事件找到处于状态 M 的块，处理器就发布“所有”信号。通过一个特殊的“或相连”网络 OR 把所有独立“所有”信号聚集起来；如果有处理器发布“所有”信号，存储器控制器就忽略该请求。注意在非流水互连中，这种特殊的网络是无关紧要的（即它是一个 OR 网关）。

然而，在高性能流水互连中，这种网络就变得很复杂了。第二种替代方法是在存储器控制器中加入一个简单的目录，它可以跟踪决定存储器控制器是否应该响应数据请求，或一个处理器节点是否应该负责这种响应。

- a. [25]<4.2, 4.4>使用一个表来指定实现第二种方法所需的存储器控制器协议。在这个问题中，请忽略 Cache 替换中的 PUTM 消息。
- b. [25]<4.2, 4.4>为支持下面的代码序列，存储器控制器必须做什么？假设 Cache 的初始内容如图 4.37 所示。

```
P1: read 110
P15: read 110
```

- 4.12 [30]<4.2>习题 4.3 将所有者状态加入简单的 MSI 监听协议。请将所有者状态加入到上面介绍的交换监听协议，并回答和习题 4.3 相同的问题。
- 4.13 [30]<4.2>习题 4.5 将独占状态加入简单的 MSI 监听协议。讨论为什么将独占状态加入到上面介绍的交换监听协议会更困难？请给出由这样的操作所产生的问题的例子。
- 4.14 [20/20/20/20]<4.6>顺序连贯性（SC）要求所有的读和写看上去是按一定全局顺序执行的。这可能要求在一定情况下，处理器在提交读或写指令之前需要停止。考虑如下的代码序列：

```
write A
read B
```

操作 write A 的结果是 Cache 缺失, 操作 read B 的结果是 Cache 命中。在 SC 下, 处理器必须使 read B 操作延时, 直到它可以安排 (并执行) write A 操作。SC 的一个简单实现是使处理器停顿, 直到 Cache 收到数据并执行写操作。

较弱的连贯性模型在对读和写操作的顺序限制上有所放松, 这就减少了处理器必须停止的情况。全局存储顺序 (TSO) 连贯性模型要求所有的写操作是按一定全局顺序执行的, 但是允许一个处理器的读操作可以提到写操作之前执行。这就允许处理器实现写 Cache, 在其中存放要执行的写操作, 这些写操作尚未和其他处理器的写操作一起进行全局排序。在 TSO 中, 允许读操作超过 (潜在的是旁路绕过) 写 Cache, 这在 SC 下是不行的。

假设一个存储器操作可以在每个时钟周期内执行, 而且当 Cache 命中或满足写 Cache 条件时不会再引入任何时钟周期停顿。缺失操作会产生如图 4.41 所列的时延。假设如图 4.37 所示的 Cache 内容, 以及如习题 4.8 所示的基本交换协议。那么对于 SC 和 TSO 连贯性模型来说, 在每个操作之前会产生多少个时钟周期停顿?

a. [20]<4.6> P0: write 110 <-- 80

P0: read 108

b. [20]<4.6> P0: write 100 <-- 80

P0: read 108

c. [20]<4.6> P0: write 110 <-- 80

P0: write 100 <-- 90

d. [20]<4.6> P0: write 100 <-- 80

P0: write 110 <-- 90

- 4.15 [20/20]<4.6> 上面提到的交换监听协议可以保证一个节点在另外的节点有可写块时不会执行读和写操作, 这是对顺序连贯性的部转移持。一个改进的协议一收到自己的 GetModified 请求就会执行写操作, 当数据消息到达时会将新写入的字数数据并入块的剩余部分中。这可能是不合法的, 因为有可能另外一个节点同时也在执行写操作。尽管如此, SC 所要求的全局顺序是由地址网络中的一致性要求所决定的, 所以在请求者的写操作之前, 其他节点的写操作需要排序。注意这种优化不改变存储器连贯性模型。

假设有如图 4.41 所示的参数:

a. [20]<4.6> 对于有序内核来说, 优化效果有多明显?

b. [20]<4.6> 对于无序内核来说, 优化效果有多明显?

范例分析 3: 简单的基于目录的一致性

通过这个范例阐明以下概念:

- 目录一致性协议的状态转换。
- 一致性协议的性能。
- 一致性协议的优化。

图 4.42 给出了一种分布式共享存储器系统。每个 Cache 都是直接映射式的, 并有 4 个双字大小的块。为了简化说明, 每个 Cache 地址块包括完全的地址, 且每个字只显示两个十六进制字节 (最低有效字在右边)。Cache 状态有 3 种: M (已修改)、S (已共享) 以及 I (无

- 4.17 [10/10/10/10]<4.4>目录协议比监听协议更具扩展性,这是因为它们向拥有块副本的节点发送明确的请求和无效消息;监听协议则是向所有的节点广播所有的请求和无效消息。考虑如图 4.42 给出的 16 处理器系统,假设所有的块都不含有无效块。对下面的每组操作,请回答哪个节点得到什么样的请求和无效消息。
- [10]<4.4> P0: write 100 <-- 80
 - [10]<4.4> P0: write 108 <-- 88
 - [10]<4.4> P0: write 118 <-- 90
 - [10]<4.4> P1: write 128 <-- 98
- 4.18 [25]<4.4>习题 4.3 将所有者状态加入简单的 MSI 监听协议。请将所有者状态加入到上面介绍的简单目录协议,并回答和习题 4.3 相同的问题。
- 4.19 [25]<4.4>习题 4.5 将独占状态加入简单的 MSI 监听协议。讨论为什么将独占状态加入到上面介绍的简单目录协议会更困难?请给出由这样的操作所产生的问题的例子。

范例分析 4: 改进目录协议

通过这个范例阐明以下概念:

- 目录一致性协议的实现。
- 一致性协议的性能。
- 一致性协议的优化。

范例分析 3 中的目录一致性协议描述了抽象层面上的一致性,并且隐含了这样一条前提假设,即对共享总线的访问是原子的。使用一条或多条流水线以及交换互连网络的高性能监听系统在极大地提高了带宽的同时,也引入了临时状态以及非原子转换等问题。目录 Cache 一致性协议比监听 Cache 一致性协议更具可扩展性,这主要是因为两方面的原因:第一,监听 Cache 一致性协议向所有的节点广播请求,这就限制了可扩展性。目录协议使用向目录发送消息的间接方法来保证请求只发送给拥有块副本的节点;第二,监听系统的地址网络必须按照一定的全局顺序来处理请求,目录协议在这方面则降低了限制要求。一些目录协议假定没有网络排序,这种假设带来了好处,它可以允许使用自适应寻径技术来改善网络带宽。其他的协议则依赖于点对点顺序(如 P0 节点到 P1 节点的消息会按顺序到达)。即使有顺序上的限制,目录协议比监听协议也具有更多的临时状态。图 4.43 给出了依赖于点对点顺序的一种简化的目录协议的状态转换。图 4.44 给出了目录控制器的状态转换。对每个块来说,目录用来保存状态和当前所占据的空间或最近的共享列表(如果有的话)。就像前面提到的高性能监听协议一样,按照表示当前状态的行和表示事件的列就可以确定一个<动作/下个状态>项。如果选中项没有新状态,那么块状态保持不变。一些不可能出现的情况被标记为“错误”,代表错误情况。“z”表示请求事件当前无法处理。

下面的例子阐述了协议的基本操作。假设一个处理器试图对一个处于状态 I(无效)的块进行读操作,相应操作为“发送 GetM/IM^{AD}”,这表示 Cache 控制器应该向目录发出 GetM(如 GetModified)请求,并转换到临时的 IM^{AD} 状态。在最简单的情况下,请求消息找到处于状态 DI 的目录,这表示任何 Cache 都不拥有副本。目录以包含预期的应答数目(本例为 0)的数据消息响应。在这个简化协议中,Cache 控制器将这个简单消息视为两个消息:数据消息及紧接着到来的最后应答事件。数据消息首先被处理,先保存数据并转换状态到 IM^A;然后处理最后应答(Last Ack)事件,将状态转换到 M。最后,在 M 状态下执行写操作。

状态	读	写	替换	INV	Forwarded_ GetS	Forwarded_ GetM	PutM _Ack	数据	Last ACK
I	发送 GetS/IS ^D	发送 GetM/ IM ^{AD}	错误	发送 Ack/I	错误	错误	错误	错误	错误
S	执行 读操作	发送 GetM/ IM ^{AD}	I	发送 Ack/I	错误	错误	错误	错误	错误
M	执行 读操作	执行写操作	发送 PutM/ MI ^A	错误	发送数据 发送 PutMS /MS ^A	发送数据 /I	错误	错误	错误
IS ^D	z	z	z	发送 Ack/ ISI ^D	错误	错误	错误	保存数据, 执行 读操作 /S	错误
ISI ^D	z	z	z	发送 Ack	错误	错误	错误	保存数据, 执行 读操作 /I	错误
IM ^{AD}	z	z	z	发送 Ack/	错误	错误	错误	保存数据, /IM ^A	错误
IM ^A	z	z	z	错误	IMS ^A	IMI ^A	错误	错误	执行写 操作 /M
IMI ^A	z	z	z	错误	错误	错误	错误	错误	执行写操 作, 发送 数据 /I
IMS ^A	z	z	z	发送 Ack/ IMI ^A	z	z	错误	错误	执行写操 作, 发送 数据 /S
MS ^A	执行 读操作	z	z	错误	发送数据	发送数据 MI ^A	/S	错误	错误
MI ^A	z	z	z	错误	发送数据	发送数据 /I	/I	错误	错误

图 4.43 广播监听 Cache 控制器的转换图

状态	GetS	GetM	PutM (所有者)	PutMS (无所有者)	PutM (所有者)	PutMS (无所有者)
DI	发送数据, 添加到共 享者 /DS	发送数据, 清除 共享者, 设置所 有者 /DM	错误	发送 PutM_Ack	错误	发送 PutM_Ack
DS	发送数据, 添加到共 享者 /DS	发送 INV 到共 享者, 清除共享 者, 设置所有者, 发送数据 /DM	错误	发送 PutM_Ack	错误	发送 PutM_Ack
DM	转发 GetS, 添加到共 享者 /DMS ^D	转发 GetM, 发送 INV 到共享者, 清除共享者, 设 置所有者	保存数据, 发送 PutM_Ack/DI	发送 PutM_Ack	保存数据, 添 加到共享者, 发送 PutM_Ack/ DS	发送 PutM_Ack
DMS ^D	转发 GetS, 添加到共 享者	转发 GetM, 发送 INV 到共享者, 清除共享者, 设 置所有者 /DM	保存数据, 发送 PutM_Ack/DS	发送 PutM_Ack	保存数据, 添 加到共享者, 发送 PutM_Ack/ DS	发送 PutM_Ack

图 4.44 目录控制器的转换图

如果GetM找到状态DS下的目录,目录就会向共享列表上的所有节点发送无效(INV)消息,并将数据发送给请求者和共享者,然后将状态转换为M。当INV消息到达共享者时,它们会发现块或者处于状态S,或者处于状态I(如果它们已经暗中使块无效)。这两种情况下,共享者会直接向请求者发送应答,请求者然后统计收到的应答数目并和其发回数据消息的块数目相比较。当所有的应答都到达后,最后应答事件发生,触发Cache转换到M状态,并允许写操作。注意有可能所有的应答都在数据消息之前到达,但是最后应答(Last Ack)事件不会发生。这是因为数据消息包括应答数目。因此本协议假定数据消息在最后应答事件之前处理。

4.20 [10/10/10/10/10/10]<4.4>考虑上面的高级目录协议和如图4.20的Cache内容。下面的每条操作会导致受影响块的状态发生怎样的转换?请分别给出状态转换顺序。

- a. [10]<4.4> P0: read 100
- b. [10]<4.4> P0: read 120
- c. [10]<4.4> P0: write 120 <-- 80
- d. [10]<4.4> P15: write 120 <-- 80
- e. [10]<4.4> P1: read 110
- f. [10]<4.4> P0: write 108 <-- 48

4.21 [15/15/15/15/15/15/15]<4.4>考虑上面的高级目录协议和如图4.42的Cache内容。受下面每组操作所影响的块,其状态转换顺序是怎样的?所有的情况都假设处理器在相同时钟周期内发出请求,但是地址网络按严格顺序为所有的请求排序。假定控制器的操作看起来是原子的(例如,目录控制器要在处理其他对相同块的请求之前,执行所有从DS到DM的状态转换所要求的操作)。

- a. [15]<4.4> P0: read 120
P1: read 120
- b. [15]<4.4> P0: read 128
P1: write 120 <-- 80
- c. [15]<4.4> P0: write 120
P1: read 120
- d. [15]<4.4> P0: write 120 <-- 80
P1: write 120 <-- 90
- e. [15]<4.4> P0: replace 110
P1: read 110
- f. [15]<4.4> P1: write 110 <-- 80
P0: replace 110
- g. [15]<4.4> P1: read 110
P0: replace 110

4.22 [20/20/20/20/20/20]<4.4>对于图4.42给出的多处理器和如图4.43及图4.44所描述的协议实现,假设有以下时延:

- CPU的读和写中不产生时钟周期停顿。
- 只有在为了响应最后应答事件而执行时,完成一个缺失(如读和写)的时延才为 L_{ack} 时钟周期;否则该操作将在数据被复制到Cache中时才完成。

- CPU 的读和写产生替换事件, 在 PutModified (如使用写回 Cache) 消息之前发出相应的 GetShared 或 GetModified 消息。
- 发送请求或应答消息的 Cache 控制器事件的时延为 $L_{\text{send_msg}}$ 时钟周期。
- 读 Cache 和发送数据消息的 Cache 控制器事件时延为 $L_{\text{send_data}}$ 时钟周期。
- 更新 Cache 和接受数据消息的 Cache 控制器事件时延为 $L_{\text{rcv_data}}$ 时钟周期。
- 存储器控制器发送请求消息的时延为 $L_{\text{send_msg}}$ 时钟周期。
- 对每个必须发送的无效信息, 存储器控制器的时延为 L_{inv} 时钟周期。
- 对每个收到的无效信息 (时延计算到发送应答消息为止), Cache 控制器的时延为 $L_{\text{send_msg}}$ 时钟周期。
- 读存储器和发送数据消息的存储器控制器事件时延为 $L_{\text{read_memory}}$ 时钟周期。
- 向存储器写数据消息的存储器控制器事件时延为 $L_{\text{write_memory}}$ 时钟周期。
- 非数据 (如请求、无效、应答) 消息的网络时延为 $L_{\text{req_msg}}$ 时钟周期。
- 数据消息的网络时延为 $L_{\text{data_msg}}$ 时钟周期。

请使用图 4.45 总结的性能特征给出一个实现的举例。

操作	实现 1
	时延
send_msg	6
send_data	20
rcv_data	15
read_memory	100
write_memory	20
inv	1
ack	4
req_msg	15
data_msg	30

图 4.45 目录一致性时延

对下面的每条操作, 使用上面的改进目录协议和如图 4.42 的 Cache 内容, 每个处理器节点观察到的时延是多少?

- [20]<4.4> P0: read 100
 - [20]<4.4> P0: read 128
 - [20]<4.4> P0: write 128 <-- 68
 - [20]<4.4> P0: write 120 <-- 50
 - [20]<4.4> P0: write 108 <-- 80
- 4.23 [20]<4.4>在 Cache 缺失的情况下, 前面介绍的交换监听协议和本范例中的目录协议都会尽快地执行读或写操作。特别是, 它们将这一操作是作为向稳定状态转换的一部分来执行的, 而不是只是向稳定状态的转变和简单的再次尝试操作。这不是优化, 而是为了保证后续处理, 协议的实现必须保证它们在放弃块之前至少执行一次 CPU 操作。
- 假设一致性协议的实现并没有做到这一点。请解释这会如何导致“活锁”现象的产生。给出一个可以模拟这种行为的简单代码例子。
- 4.24 [20/30]<4.4>一些目录协议加入了所有者状态 (通常标示为“O”), 这和前面讨论的监听协议的优化相似。当节点只能读所有者状态的块时, 所有者状态和共享状态类似; 而当其他

节点访问所有者状态的块产生读和写缺失时,节点必须提供数据,则所有者状态和修改状态相似。在原来的协议中,对一个处于 Modified 状态块的 GetShared 请求要求节点向请求处理器和存储器都发送数据,所有者状态的引入消除了这种情况的发生。在 MOSI 目录协议中,对 Modified 或所有者状态块的 GetShared 请求要向请求节点提供数据,并转换到所有者状态。对所有者状态的 GetModified 请求的处理和 Modified 状态的请求处理相似。当节点取代所有者状态或修改状态的块时,这种优化的 MOSI 协议只更新存储器。

a. [20]<4.4>请解释为什么协议中的 MS^A 状态是重要的“临时”所有者状态。

b. [20]<4.4>请对 Cache 和目录协议表进行修改,以支持稳定的所有者状态。

4.25 [25/25]<4.4>上面介绍的改进目录协议依赖于点对点的顺序互连来保证正确的操作。假设 Cache 的开始状态如图 4.42 所示,对于下面的每组操作,解释如果互连不能维持点对点的顺序会出现什么问题。假定处理器在同一时间内提出请求,但是请求是按照目录中的顺序被处理的。

a. [25]<4.4> P1: read 110

P15: write 110 <-- 90

b. [25]<4.4> P1: read 110

P0: replace 110

第5章 存储器层次结构设计

在理想情况下,我们希望存储容量无限大,这样,任何一个特定的……字都可以立刻得到……我们……不得不意识到构建存储器层次结构的可能性,它们中的每一层都比其上一层具有更大的容量,但访问速度却慢一些。

A. W. Burks, H. H. Goldstine 和 J. von Neumann
Preliminary Discussion of the Logical Design of
an Electronic Computing Instrument (1946)

5.1 简介

程序员对存储器容量和速度的期望是无限的,关于这一点计算机的先驱者们早就正确预见到了。而解决这个问题的一种较经济的方法,就是利用局部性原理和存储器成本性能分析技术,采用存储器层次结构。在第1章中所阐述的局部性原理认为大部分程序并不是均衡地访问所有的代码和数据。局部性包括时间局部性和空间局部性。局部性原理和“越小的硬件速度越快”的原则,共同导致了基于不同速度和容量的存储器层次结构的产生。图5.1给出了一个包括典型的容量和访问速度的多级存储器层次结构。

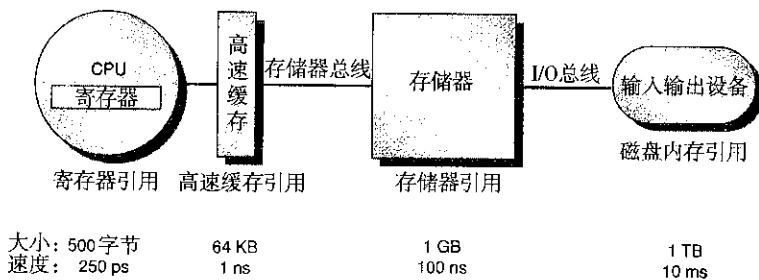


图 5.1 一个适用于嵌入式系统、台式机和服务器的通用典型多级存储器层次结构。距离处理器越远的层次,存储容量越大,访问速度越慢。注意,不同层次之间的时间单位在 10^{-12} 秒 (ps) 到 10^{-3} 秒 (ms) 之间变化;而容量单位在几个字节到 10^{12} 字节 (TB) 之间变化

由于快速存储器价格昂贵,存储器层次结构被组织成不同的层次——每一层都比其下一层容量更小,速度更快,每字节成本也更高。这样做的目标是提供一个存储系统,使之能够具有几乎相当于最便宜层次的存储器的价格,但是访问速度却与最快层次的存储器接近。

请注意,每一层都要从一个较大的存储器空间中把地址映射到一个较小但是更快的上层存储器。作为地址映射的一部分,存储器层次结构应该进行地址检查,因此,用于检查地址的保护机制也是存储器层次结构的一部分。

存储器层次结构的重要性随着处理器性能的提高而不断增加。图 5.2 指出随着存储器性能的提高, 处理器的性能飞速增长的发展过程。很明显, 计算机系统结构设计者必须尽力去缩小处理器-存储器之间的性能差距。

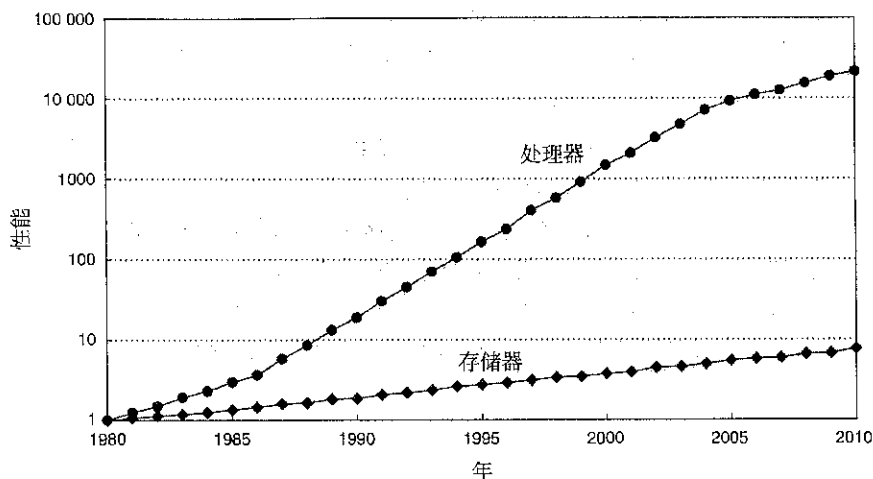


图 5.2 以 1980 年的性能为基准, 存储器和处理器的性能差距随时间的变化曲线。注意, 纵轴必须用对数坐标来表示处理器-DRAM (存储器) 性能差距的大小。存储器的基准是 1980 年的 64 KB DRAM, 在时延方面的性能增长为年均 7% (见图 5.30)。处理器曲线表示到 1986 年为止处理器性能每年增长 1.25 倍, 而此后到 2004 年每年增长 1.52 倍, 2004 年以后年增长率变为 1.20 倍, 参见图 1.1

随着存储器容量的增加, 以及这种性能差距的影响日益显著, 存储器层次设计的基本概念现在已经出现在大学本科有关计算机系统结构的课程中了, 甚至有关操作系统和编译的课程对此也有所涉及。所以在这里, 我们对 Cache 只做简单的回顾, 其余大部分内容将主要描述针对处理器-存储器之间的性能差距的更多高级改进方法。

当要读取的字不在 Cache 中时, 它首先必须从存储器中取出, 然后放置到 Cache 中去。对于多个字组成的块 (或线), 出于效率的考虑, 会对其进行移动操作。每个 Cache 块包含一个标签, 以指示相应的存储器地址。

就设计而言, 最关键的问题是确定在 Cache 中放置块的位置。应用最广泛的策略是组相联 (set associative), 一个组 (set) 是 Cache 中的一组块。一个块首先被映射到一个组中, 然后它可以被放置到组中的任何一块中。查找一个块, 首先从块映射到组的地址, 然后在组内并行查找, 直至找到该块。组通常利用数据的地址来确定, 即

$$(\text{块地址}) \bmod (\text{Cache 中的组数})$$

如果一个组中有 n 块, 那么这个 Cache 的映射方法就称为 n 路组相联。 n 路组相联分别有如下两种极端情况: 直接映射仅仅是一个简单的 1 路组相联 (块总放置在同样的位置); 全相联是只有 1 组的组相联映射 (块能放置在 Cache 中的任何位置)。

对 Cache 中数据的读操作很容易, 因为在 Cache 中的副本和存储器中的相应内容是一样的。相对而言, Cache 的写操作就困难些: 如何使 Cache 中的副本和存储器保持一致? 有两种主要的策略: 写直达 (write-through) —— 信息被同时写到 Cache 块和存储器块中; 写回法 (write-back) —— 信息只被写入 Cache 块中, 只有 Cache 中的块将被替换时, 该信息才会被写回到存储器中。两种写操

作策略都能使用一个写缓冲区,一旦数据进入写缓冲区,就允许 Cache 对它进行操作,而不用等待将数据写入存储器中的全部时延。

一种评价不同 Cache 组织结构优劣的方法是缺失率。缺失率可以简单地理解为 Cache 访问的分片造成的缺失,即访问失败的次数和总访问次数的比值。

为对高缺失率的原因有更深入的研究,从而更好地设计 Cache,把所有缺失归结为如下的 3C 模型:

- **强制 (Compulsory)**: 对块的第一次访问一定不在 Cache 中,所以该块必须被调入到 Cache 中。即使有无限大的 Cache,强制缺失也会发生。
- **容量 (Capacity)**: 如果 Cache 容纳不了一个程序执行所需要的所有块,将会发生容量缺失 (另外还有强制缺失),某些块将被放弃,随后再被调入。
- **冲突 (Conflict)**: 如果块未采用全相联放置策略,冲突缺失 (另外还有强制和容量缺失) 将会发生,如果有太多的冲突块映射到同一组中,产生冲突,则某一个块可能被放弃随后再调入。

图 C.8 和图 C.9 显示了按照上述 3C 模型划分的 Cache 缺失频率 (第 4 章加入了第 4 个 C,即一致 (coherency) 缺失,由于 Cache 刷新而保持在多处理器中的多 Cache 一致性,这里我们不考虑)。

不过,缺失率也会由于某些原因而产生误导。因此,有些设计者选用每条指令的缺失次数 (misses per instruction) 而不是每次存储访问的缺失数 (misses per memory reference) 来度量缺失率。两者有相关之处:

$$\frac{\text{缺失率}}{\text{指令数}} = \frac{\text{缺失率} \times \text{存储器访问次数}}{\text{指令数}} = \text{缺失率} \times \frac{\text{存储器访问次数}}{\text{指令数}}$$

为避免使用小数,通常用整数形式的每 1000 条指令缺失数来表示缺失。对预测处理器而言,仅仅统计已提交的指令。

上述两种评价均存在没有考虑缺失代价的问题。一种更好的评价方法是使用访问存储器的平均时间:

$$\text{存储器的平均访问时间} = \text{命中时间} + \text{缺失率} \times \text{缺失代价}$$

这里的命中时间 (hit time) 是 Cache 命中的时间,缺失代价是由于访问目标不在 Cache 中而从存储器中替换该块所花费的时间。平均访问时间仍然不是一个直接的性能评价方法;虽然用它比用缺失率更好,但它不能代替执行时间。例如,在第 2 章中,我们看到预测处理器会在缺失期间执行其他的指令,从而有效地降低缺失代价。

初学者请参看附录 C。它给出了更多更深入的介绍,同时包含了实际计算机的 Cache 实例和其 Cache 效率定量的评估。

附录 C 中的 C.3 小节也介绍了 6 种基本的 Cache 优化方法,我们将在此简单回顾一下。附录将会给出这些优化实例的定量分析。

1. **增加块容量来降低缺失率**: 最简单的降低缺失率的办法是利用空间局部性原理,增加块容量。需要注意的是,过大的块在降低强制缺失的同时,也会增加缺失代价。
2. **增大 Cache 来降低缺失率**: 最显而易见的降低容量缺失的方法是增大 Cache 容量。大容量 Cache 潜在的缺点是命中时间较长,而且价格开销和功耗都较大。

3. 增加相联度来降低缺失率：增加相联度能明显地降低冲突缺失，但较大的相联度也会造成命中时间增大。
4. 使用多级 Cache 来降低缺失率：应该使 Cache 的命中时间更快，达到与 CPU 一样的速度？还是应该使 Cache 足够大以弥合 CPU 和存储器之间的不断加大的性能差距？人们一直以来很难在这两者之间做出抉择。通过在原始 Cache 和存储器之间增加另一级 Cache 能使这个问题得到简化（见图 5.3）。第一级 Cache 就可以小到足以跟上快速的 CPU 的时钟周期，而第二级 Cache 能够大到足以支持对存储器进行的大多数访问。对第二级 Cache 缺失的研究导致了更大的块、更大的容量和更高的相联度的出现。用 L1 和 L2 来分别表示第一级和第二级 Cache，则可以重新定义平均存储器访问时间：

$$\text{命中时间}_{L1} + \text{缺失率}_{L1} \times (\text{命中时间}_{L2} + \text{缺失率}_{L2} \times \text{缺失代价}_{L2})$$

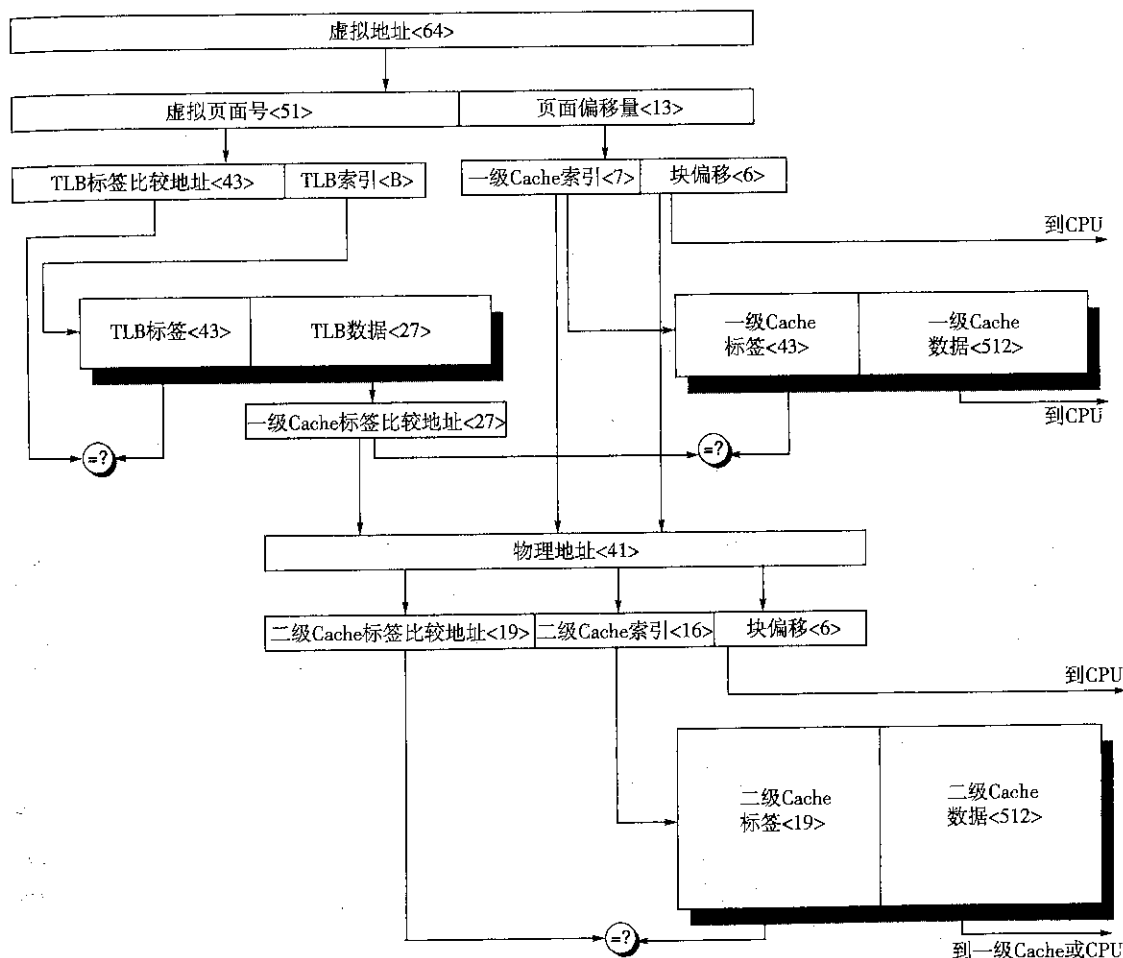


图 5.3 一个从虚拟地址到二级 Cache 的存储器层次结构图。页面大小是 8 KB。TLB 直接映射 256 个条目。直接映射的一级 Cache 是 8 KB，直接映射的二级 Cache 是 4 MB，都是用 64 字节的块。虚拟地址是 64 位，物理地址是 40 位。此图与实际存储器层次结构的主要区别是 Cache 和 TLB 相联程度更高，以及比 64 位更小的虚拟地址，如图 5.18 所示

5. 使读缺失的优先级高于写操作来降低缺失代价：在写缓冲区中可以实现此优化。因为写缓冲区拥有读缺失所需要的每个位置的更新值，所以产生了冒险，即在存储器中RAW的冒险。一种解决方法是，在读缺失时检查写缓冲区的内容。如果没有冲突或者存储系统可用，则在写之前进行读操作，以降低缺失代价。大部分处理器中读操作的优先级高于写操作。
6. 在 Cache 索引过程中避免地址变换以降低命中时间：Cache 需要应对从处理器的虚拟地址到存储器物理地址的变换（虚拟存储器将在 5.4 节和 C.4 节中讲到）。一个典型的 Cache、变换旁视缓冲器（TLB）或快表和虚拟存储器之间的关系如图 5.3 所示。通用的优化方法是使用页面偏移地址来索引 Cache，在虚拟地址和物理地址中，这部分地址是一致的。如果使用虚拟地址来索引 Cache 表项，那么用物理地址来确认是否命中。Cache 在判断和存取时不做虚拟地址到物理地址的转换，速度可以更快。但这种“虚拟地址索引、物理地址确认”优化的缺点是页面的大小限制了 Cache 的容量。例如，直接映射的 Cache 容量不能超过页面大小。更高的相联度可用物理地址来索引 Cache，同时支持超过页面大小的 Cache 容量。例如，组相联度加倍的同时 Cache 大小也加倍，而索引域宽度保持不变。因为索引域宽度由如下公式决定：

$$2^{\text{Index}} = \frac{\text{Cache 大小}}{\text{块大小} \times \text{组相联度}}$$

从表面上看，显然可以使用虚拟地址作为替代办法来访问 Cache，但这会造成操作系统额外的开销。

注意，上述 6 种降低平均存储器访问时间的优化方法，都有导致平均存储器访问时间增加的可能性。

本章中剩下的内容要求读者已经对包括图 5.3 在内的上述内容有所了解。为和应用相结合，全章（包括附录 C）将以 AMD Opteron 微处理器的存储层次结构为例来讲述 Cache 的思想。在章节的末尾，我们使用 SPEC2000 基准测试程序，对层次结构对性能的影响进行评估。

Opteron 是一款为桌面系统和服务器设计的微处理器。但这两款不同种类的计算机对存储层次结构有不同的要求。桌面型计算机常常是单个用户在操作系统之上一次运行一个应用，反之，服务器系统可能同时有多个用户运行多个应用。这些特性导致存储器中的内容频繁变化，而增加了丢失率。因此，桌面型计算机关注的是存储层次结构中的平均时延，而服务器系统除此之外还要关注存储带宽。

5.2 11 种先进的 Cache 性能优化方法

上节给出的存储器平均访问时间的公式提供了 Cache 优化的三个参数：命中时间、缺失率和缺失代价。对于超标量处理器的广泛使用，我们还加入了 Cache 带宽。这样，可以把 11 种先进的 Cache 性能优化方法分成以下几类：

- 减少命中时间：小而简单的 Cache、路预测以及踪迹 Cache。
- 增加 Cache 带宽：流水线 Cache、多组 Cache 和非阻塞 Cache。
- 降低缺失代价：关键字优先及合并写缓冲区。
- 降低缺失率：编译器优化。
- 通过并行降低缺失代价和缺失率：硬件预取和编译器预取。

我们将在图 5.11 中，对上述 11 种性能优化技术的实现复杂性给出简要总结。

第一种优化：小而简单的 Cache 减少命中时间

Cache命中时间中最耗时的部分是按地址的索引字段去读标志存储器，然后再与地址比较。较小的硬件速度更快，因此小Cache当然有助于减少命中时间。此外，关键的一点是，应该使二级Cache也应尽可能小，这样它就能与处理器放在同一个芯片里面，从而避免片外时间代价。

第二个建议是保持Cache的简单性，比如采用直接映射方式。采用直接映射Cache可以使标志检查与数据传输并行，有助于减少命中时间。

因此，要实现快速时钟周期，就需要将第一级Cache做得尽可能小而简单。对于更低层次的Cache，一些设计者为了在提高标志检查速度的同时提供一个容量的独立存储器芯片，采用了一种折中方案，即将标志放在片内，而将数据放在片外。

尽管在新一代微处理器中，片内Cache的数量在增加，但一级Cache的容量并未增加。以AMD的三代处理器K6, Athlon和Opteron为例，它们的一级Cache的容量是相同的。人们把重点放在了在快速时钟频率下，通过动态执行隐藏一级Cache缺失和使用二级Cache来避免直接访问存储器。

在设计芯片之前可以使用CAD工具来确定对命中时间的影响。CACTI程序是一个可以评估CMOS微处理器各种Cache结构访问时间的CAD工具。对于一个给定的最小特征值，可以改变Cache容量、相联度和读/写端口的数目，以估计各种情况时的Cache命中时间。图5.4反映了各种Cache容量和相联度的命中时间值。根据Cache面积大小的不同，图中命中速度参数显示：直接映射比2路组相联快1.2~1.5倍；2路组相联比4路组相联快1.02~1.11倍；4路组相联比全相联快1.0~1.08倍。

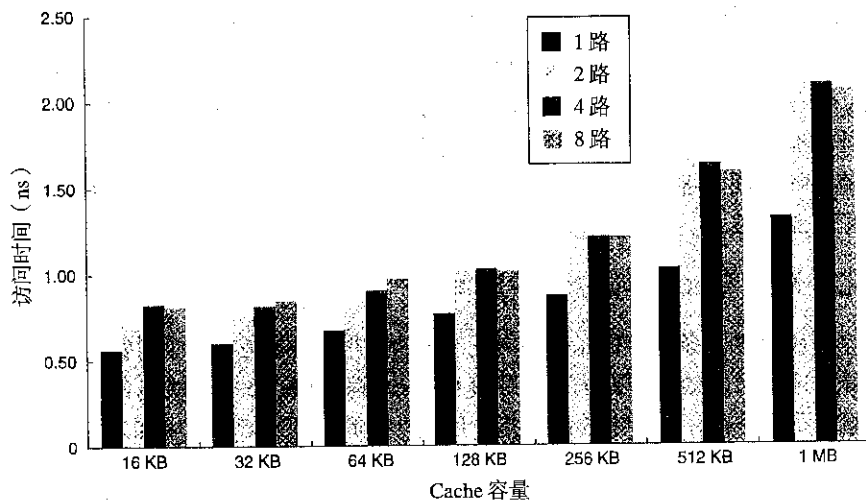


图 5.4 CMOS Cache 中不同容量和相联度下的访问时间比较。数据是由 Tarjan, Thoziyoor 和 Jouppi[2006]在 CACTI 模型 4.0 上取得的。Cache 中使用的参数如下：90 nm 工艺，一个单读/写端口，64 位字节块。其中 2 路、4 路、8 路组相联相对于直接映射的访问时间的比值分别为 1.32, 1.44 和 1.52

例题 假设一个 2 路相联的一级数据 Cache 比一个同样大小的 4 路相联的 Cache 的命中时间快 1.1 倍。如附录 C 中的图 C.8，每 8 KB 数据的缺失率从 0.049 下降到 0.044。假设 Cache 正好在时间的临界值上，1 个时钟周期 1 次命中。假设 2 路组相联的 Cache 对二级 Cache 的缺失代价是 10 个时钟周期，且二级 Cache 不发生缺失。哪种方式的平均访存时间快？

解答: 对2路组相联Cache:

$$\begin{aligned}\text{平均访存时间}_{2\text{路}} &= \text{命中时间} + \text{缺失率} \times \text{缺失代价} \\ &= 1 + 0.049 \times 10 = 1.49\end{aligned}$$

对4路组相联Cache, 时钟周期比2路组相联长1.1倍。因为不受处理器时钟频率的影响, 缺失代价的持续时间相同, 故假设它花了9个较长的时钟周期:

$$\begin{aligned}\text{平均访存时间}_{4\text{路}} &= \text{命中时间} \times 1.1 + \text{缺失率} \times \text{缺失代价} \\ &= 1.1 + 0.044 \times 9 = 1.50\end{aligned}$$

如果时钟周期真的延长到1.1倍, 那么这对性能的影响甚至比存储器平均访问时间还要大, 因为即使处理器不访问Cache, 时钟频率也会变慢。

尽管有这些优势, 由于许多处理器至少需要2个时钟周期来访问Cache, 因此一级Cache如今至少还是2路组相联的。

第二种优化: 路预测减少命中时间

这是另外一种减少冲突缺失同时又能确保直接映射Cache命中速度的方法。在路预测中, 需要在Cache中预留特殊的比特位, 用来预测下一次访问Cache时可能会在组中用到的路或块。这种预测意味着提前使用多路选择器来选择所需的块, 而且在读缓存数据时, 该时钟周期内只需比较一个标志位。如果缺失, 则在接下来时钟周期中检查其他的块是否匹配。

附加到Cache中各个块上的是块预测位。它决定在下次访问Cache时优先选择哪些块。如果预测器成功, Cache访问时延就是最快的命中时间。如果失败, 它将选择其他的块, 改变路预测器会有一个额外的时钟周期时延。模拟表明对于2路组相联来说, 组预测的精度有85%, 故路预测节省了85%的流水线时间。路预测很适合推测处理器, 因为在预测失败时操作已经被取消。Pentium 4处理器使用了路预测技术。

第三种优化: 踪迹Cache减少命中时间

对于每个时钟周期多条指令的指令级并行, 需要解决的问题是在每个时钟周期能否提供足够的无关指令。解决这个问题的一种方法就是踪迹Cache。踪迹Cache中的块包含的是对已执行指令的动态跟踪信息, 而不是在存储器中的静态指令序列。因此需要在Cache中加入转移预测, 在进行一次合法操作时, 它必须和地址一起验证。

显然, 踪迹Cache的地址变换机制更加复杂, 因为地址不再是2的 n 次幂倍字长对齐的(n =字长)。然而, 它们更好地利用了指令Cache中的长块。传统Cache中的长块允许转移从中间进入和到达末端之前退出, 因此它们具有极差的空间利用率。踪迹Cache的缺点是, 具有不同选择的条件转移导致了相同的指令成为不同跟踪的组成部分, 每一份副本在踪迹Cache中都占据了一定的空间, 从而降低了空间利用率。

注意, 在Pentium 4处理器的踪迹Cache中使用译码微操作, 由于节省了译码时间, 它给出了另外一种性能优化的思路。

许多优化技术容易理解且应用广泛, 但踪迹Cache却既不简单也不常用。比起它的优势, 其成本昂贵, 功耗和复杂度都很高。因此, 踪迹Cache可能只是一种昙花一现的方法。这里之所以提及它, 是因为在流行的Pentium 4处理器中在这方面有所应用。

第四种优化：流水线 Cache 访问

此方法将流水线、Cache 访问、使一级 Cache 命中时的有效时延分散到几个时钟周期。它提供了较短的周期时间和高带宽，但是命中时间较长。比如，Pentium 访问指令 Cache 需要 1 个时钟周期，而 Pentium Pro 至 Pentium III 需要 2 个时钟周期，Pentium 4 需要 4 个时钟周期。存储器访问过程的流水段数增加，导致预测失败以及发出 load 指令到数据被使用之间有更多的时钟周期代价（见第 2 章）。

第五种优化：利用非阻塞 Cache 增加 Cache 带宽

对允许乱序执行的流水线机器（见第 2 章）来说，处理器是不需要在 Cache 缺失时停顿的。例如，处理器在等待数据 Cache 返回数据的同时，可以继续从指令 Cache 中取指令。非阻塞或是无锁（lockup-free）Cache 在缺失时继续保持数据 Cache 命中，使这个方案的潜在优势得到加强。这个“缺失仍命中”优化策略通过在缺失时发挥作用而不是忽略处理器请求的做法，减少了缺失代价。在这里，如果 Cache 能重叠多个缺失，可能会进一步降低实际的缺失代价，即“多重缺失仍命中”或是“缺失时仍缺失”优化策略。第二个选择仅仅在存储器系统能服务于多重缺失的情况下才能发挥作用。

图 5.5 给出了一个 8 KB 的数据 Cache 在当前缺失数量变化时，Cache 缺失所需的平均时钟周期数。在越复杂的情况下浮点程序的效果越明显，而定点程序的性能改进几乎均来自于简单的“1 次缺失时仍命中”。由第 3 章的讨论可知，同时发生多次缺失会限制程序的指令级并行性。

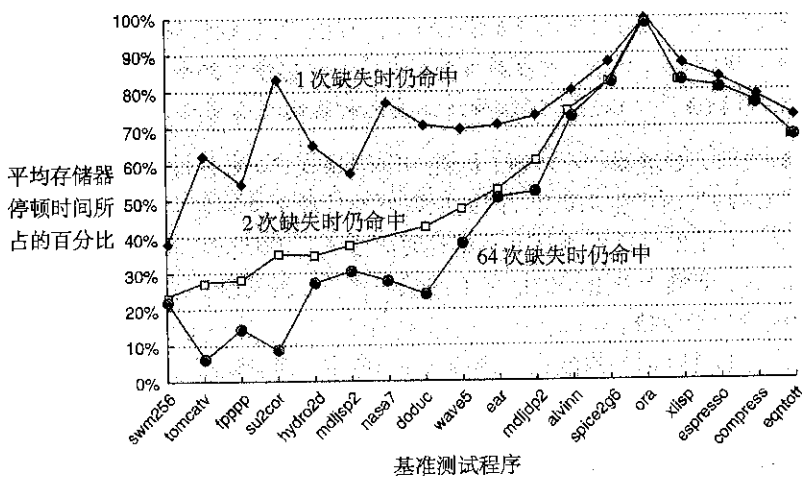


图 5.5 当 18 个 SPEC92 程序的当前缺失数量变化时，采用缺失时仍命中策略的阻塞 Cache 平均存储器停顿时间比率。64 次缺失时仍命中对应的折线允许机器中的每一个寄存器有一次缺失。前 14 个程序是浮点程序：使用 1 次缺失时仍命中策略得到的平均比率为 76%，2 次缺失时仍命中的平均比率是 51%，64 次缺失时仍命中的平均比率是 39%。最后四个定点程序，对应的三个平均比率分别为 81%，78% 和 78%。这些数据是通过一个 8 KB 直接映射的数据 Cache 采集的，它的块容量为 32 字节，缺失代价为 16 个时钟周期。这些数据来自 VLIW 的多流编译器，它对取指令和使用该数据的指令分开调度[Farkas and Jouppi 1994]。尽管这是一级 Cache 到二级 Cache 缺失的一个有效模型，但是用 SPEC2006 基准测试程序和有关缺失代价的新方法重现该实验仍然很有意义。

例题 对于图 5.5 所描述的 Cache，使用哪种策略对于浮点程序更重要呢：2 路组相联还是一次缺失命中？对于定点程序呢？假定 8 KB 数据 Cache 的平均缺失率如下：使用直接映射 Cache 的浮点程序是 11.4%，使用 2 路组相联 Cache 的浮点程序是 10.7%，使用直接映射 Cache 的定点程序是 7.4%，而使用 2 路组相联 Cache 的定点程序是 6.0%。假定平均存储器停顿时间是缺失率和缺失代价的乘积。

解答：图 5.5 中的数据假定二级 Cache 的缺失代价为 16 个时钟周期。虽然对于缺失代价来说这是比较低的，但这里为了保持一致我们还是采用这个假设。对于浮点程序平均存储器停顿时间是

$$\text{缺失率}_{\text{直接映射}} \times \text{缺失代价} = 11.4\% \times 16 = 1.84$$

$$\text{缺失率}_{2\text{路}} \times \text{缺失代价} = 10.7\% \times 16 = 1.71$$

所以说，采用 2 路组相联 Cache 的平均存储器停顿时间是采用直接映射 Cache 的平均存储器停顿时间的 1.71/1.81，即 93%。图 5.5 的标题说明 1 次缺失时仍命中把平均存储器停顿时间减少到阻塞式 Cache 的 76%，所以，对于浮点程序而言，支持 1 次缺失时仍命中的直接映射数据 Cache，能够比在缺失时阻塞的 2 路组相联 Cache 有更好的性能。

对于定点程序计算结果是

$$\text{缺失率}_{\text{直接映射}} \times \text{缺失代价} = 7.4\% \times 16 = 1.18$$

$$\text{缺失率}_{2\text{路}} \times \text{缺失代价} = 6.0\% \times 16 = 0.96$$

所以采用 2 路组相联 Cache 的平均存储器停顿时间是采用直接映射 Cache 的平均存储器停顿时间的 0.96/1.18，即 81%。图 5.5 的标题说明 1 次缺失时仍命中把平均存储器停顿时间减少到正常阻塞 Cache 的 81%，所以对于定点程序，这两种策略具有同样的性能。

由于 Cache 缺失时，处理器并不停顿，因此对非阻塞式 Cache 的性能进行评价比较困难。这种情况下很难判断单次缺失所产生的影响，因而也就更难计算平均存储器访问时间。实际缺失代价并不是所有缺失之和，而是处理器停顿这一不重叠时间之和。评价非阻塞 Cache 性能的优劣是一件复杂的工作，因为它不仅依赖于多重缺失情况下的缺失代价，还依赖于存储器引用模式，以及处理器在缺失发生时能执行的指令数。

通常，乱序执行的处理器如果能在二级 Cache 中命中，则能隐藏一级数据所产生的 Cache 缺失代价，但是却不能隐藏二级 Cache 的缺失代价。

第六种优化：利用多组 Cache 增加 Cache 带宽

与其把 Cache 看做是一个单个集成块，到不如将其划分为若干个独立的存储体以支持并发访问。划分存储体最初是用于改善存储器性能的，现在也用于 DRAM 芯片和 Cache 中。AMD Opteron 的二级 Cache 有两个存储体，Sun 的 Niagara 有四个存储体。

当按序访问存储体时，存储体工作的状态最佳，因此存储体地址映射会影响存储器系统的行为。一个简单而有效的映射是按序扩展存储体块地址，即顺序交叉。例如，有四个存储体交叉编址，存储体 0 中每个块的地址模 4 为 0，存储体 1 中每个块的地址模 4 为 1，依此类推。图 5.6 表示了这种交叉存储的编址。



图 5.6 使用块编址的 4 路交叉 Cache 存储体。假设按每个块 64 字节，则每个地址都要乘 64 来得到字节地址

第七种优化：关键字优先和提前重启动以减小缺失代价

此技术来源于经验数据：处理器在同一时刻只需要块中的一个字。这种技术不必等到全部块装入就将所需字送出，然后重新启动处理器。具体的两种策略如下：

- **关键字优先**：首先向存储器请求缺失的字，一旦它到达就把它发送给处理器，使得处理器在装入块的其他字的同时能够继续执行。
- **提前重启动**：以正常顺序获取字，块中所需字一旦到达，就立即把它发送给处理器，使处理器继续执行。

通常这些技术只适用于较大 Cache 块的情况。注意，在填充剩余的块时，Cache 通常还会继续支持其他块的访问。

依据空间局部性原理，接下来访问块中剩余部分的可能性很大。正如非阻塞 Cache 一样，这种优化方法的缺失代价也不易计算。在关键字优先中，当发生第二次请求时，有效的缺失代价是从本次缺失发生起到第二次块中其他字到达为止的不相重叠时间。

例题 假定一个计算机的处理器 Cache 块的大小为 64 字节，二级 Cache 用 7 个时钟周期获取 8 个字节的關鍵字，获取块中其他部分每 8 字节需 1 个时钟周期（这些参数如同 AMD Opteron）。首先，不采用关键字优先，开始的 8 个字节需要 8 个时钟周期，块的其他部分中每 8 个字节需要 1 个时钟周期。假定在全部块被完全装入前没有对块内其他字进行访问，首先计算关键字优先的平均缺失代价，然后计算从块的其他部分中顺序读取 8 个字节数据的平均缺失代价。比较采用关键字优先和没有采用关键字优先时所消耗的时间。

解答：采用关键字优先策略，平均缺失代价为 7 个时钟周期。不采用关键字优先策略，连续读取一整块的内容，处理器要花费 $8 + (8 - 1) \times 1 = 15$ 个时钟周期。这样，对于读一个字来说，结果是 15 比 7 个时钟周期。Opteron 每个时钟周期能发出两条 load 指令，故需要 $8/2 = 4$ 个时钟周期执行 load。不采用关键字优先，总共需要 19 个时钟周期完成 load 和 read。采用关键字优先，由于 load 是重叠进行的，总共需要 $7 + 7 \times 1 + 1 = 15$ 个时钟周期。对于读一整块内容来说，结果是 19 比 15 个时钟周期。

正如上例所示，关键字优先和提前重启动的效果依赖于块的大小以及对块中没有被载入部分的访问的可能性。

第八种优化：合并写缓冲区以降低缺失代价

因为所有的存储操作必须放到更低一级的存储层次上，因此采用写直达的 Cache 需要一个写缓冲区。即使是写回式 Cache 在替换块时也要用到一个简单的写缓冲区。如果写缓冲区为空，从处理器的角度来看，将被替换掉块的数据和地址放到写缓冲区中，写操作就已结束；写缓冲区准备将数

据写回存储器时,处理器可以继续工作。如果写缓冲区里还有其他被修改过的数据,应检查新数据的地址是否与写缓冲区中的某一合法项地址相匹配。如果匹配,新数据就合并到该项中。此优化方法称为写合并。Sun 的 Niagara 就是采用写合并技术的一种处理器。

如果缓冲区已满,并且没有地址可以匹配时,Cache (和处理器) 必须等待,直到缓冲区空出一个项。因为一次写多个字比一次写一个字的速度更快,这种优化更有效地利用了存储器。Skadron 和 Clark[1997]指出在一个 4 项的写缓冲区中,有大约 5%~10% 的性能损失是由停顿造成的。

这种方法同时减少了因缓冲区已满而产生的停顿。图 5.7 给出了采用“写合并”和未采用“写合并”的写缓冲中的情况。图中假定写缓冲区有 4 个项,每一个项有 4 个 64 位字。未采用优化时,因为一个项只能保存一个字,对连续地址的四次存储操作就会将缓冲区写满;而采用优化的写缓冲区则将这四个字合并到一个项中。

写地址	V	V	V	V
100	1	Mem[100]	0	0
108	1	Mem[108]	0	0
116	1	Mem[116]	0	0
124	1	Mem[124]	0	0

写地址	V	V	V	V
100	1	Mem[100]	1	Mem[108]
	0		0	
	0		0	
	0		0	

图 5.7 写合并图例, 图中上半部分的缓冲区未用写合并, 下半部分采用了写合并优化。写合并将四次写操作合并到缓冲区的一项中; 不采用写合并时, 尽管缓冲区每一项的 3/4 都被浪费掉, 这四次写操作还是会填满写缓冲区。缓冲区有 4 个项, 每个项包括 4 个 64 位字, 项地址在图的左边, 有效位“V”指明该项的后续 8 个字节是否已经被占用 (不采用写合并时, 图的上半部分中的每一项右边的几个字只被一次写多个字的指令所使用)

注意, 输入/输出设备的寄存器通常是直接映射到物理地址上的。这些 I/O 地址不能进行写合并, 因为分开的 I/O 寄存器操作与存储器中字的数组是不同的。例如, 每个寄存器分别需要一个地址和数据字, 而不是只使用一个地址写多个字。

在写回法中, 替换块也常称为牺牲块。故 AMD Opteron 处理器把写缓冲区称为牺牲缓冲区。写牺牲缓冲区或牺牲缓冲区包含 Cache 中因为缺失而被丢失的脏块。发生缺失时, 在访问下层存储器之前, 先检查牺牲缓冲区, 看其中是否包含所需的数据, 而不是延迟到下一次 Cache 缺失。这更像另外一种优化技术“牺牲 Cache”。与之对比, 牺牲 Cache 包含 Cache 中因为缺失而被丢失的所有块, 不管它是否是脏块[Jouppi 1990]。

写缓冲区的目的是允许 Cache 继续工作而无须等待脏块写回主存储器, 而牺牲 Cache 的目标是为减小冲突缺失的影响。尽管“牺牲”这一名称的使用造成了概念上的一些混淆, 但在目前写缓冲区还是比牺牲 Cache 要应用得更为广泛。

第九种优化：编译器优化以降低缺失率

上述优化技术都需要改变硬件。接下来降低缺失率的技术，无须修改任何硬件。

这种方法来自于软件优化——硬件设计者最满意的解决方案。处理器和存储器之间不断增大的性能差距，促使编译器设计者对存储器层次结构进行深入研究，他们试图通过编译时间的优化来改善性能。研究分为指令缺失性能改善和数据缺失性能改善两个方面。下面的优化技术在很多编译器中均有应用。

程序代码和数据重组

程序代码可以在不影响正确性的前提下方便地重新组织。例如，通过对程序中的过程重排序，可能会减少冲突缺失，从而降低指令缺失率[McFarling 1989]。另外，优化程序还能提高大容量Cache块的效率。对块进行排序，使项起始点在Cache块的起始位置，就可以减少对连续代码操作时Cache的缺失。如果编译器知道转移可能发生，它就能通过改变转移的执行，将转移目标上的基本块和该转移后的基本块的位置互换，从而改善空间局部性。这种优化称为**转移校正**。

数据不像代码那样，对存储位置有那么多的限制。这种转换操作的目的是尽力改进数据的空间和时间局部性。例如，数组计算（在科学计算中导致大部分缺失的因素）可以转换为对一个Cache块中所有数据的操作，而不是盲目地按照程序员定义的循环顺序随意对数组进行操作。

为了理解这种类型的优化，我们给出两个例子，对C代码进行手工变换来减少冲突缺失。

循环交换

一些程序带有嵌套循环，它们访问存储器中的数据是不按顺序的。简单的交换嵌套循环可以使代码按照存储顺序来访问数据。假定开始时数组不在Cache中，这种技术通过增加空间局部性来减少缺失；通过代码的重新排序，可以在块被替换之前最大限度地利用Cache块中的数据。

```
/* Before */
for (j = 0; j < 100; j = j+1)
    for (i = 0; i < 5000; i = i+1)
        x[i][j] = 2 * x[i][j];

/* After */
for (i = 0; i < 5000; i = i+1)
    for (j = 0; j < 100; j = j+1)
        x[i][j] = 2 * x[i][j];
```

原先的代码会以100个字为步长跳跃式地访问存储器，而修改后的版本在访问了一个Cache块中的所有字后才去访问下一个Cache块。这种优化策略在不影响执行指令数的前提下提高了Cache的性能。

分块

这种优化措施通过提高时间局部性来减少缺失。我们还是以多个数组为例，有些是按行访问，有些是按列访问。由于在每一次循环中行和列都被用到，因此，按行来存储数组（**行主序**）或是按列来存储数组（**列主序**）并不能解决问题。这种正交的访问意味着变换（如循环交换）仍有很大的改善空间。

分块算法并不对数组中的整行或是整列进行操作，它对子矩阵或矩阵块进行操作。其目的是在调入到Cache中的块被替换之前最大限度地利用它。下面的矩阵乘法代码是一种优化措施：

```
/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
```

```

{r = 0;
for (k = 0; k < N; k = k + 1)
    r = r + y[i][k]*z[k][j];
x[i][j] = r;
};

```

两个内层循环读取了矩阵 z 的所有 $N \times N$ 个元素，并对矩阵 y 一行中的 N 个元素进行重复访问，然后对矩阵 x 一行中的 N 个元素进行写操作。图 5.8 是一个访问 3 个数组的示意图。黑色阴影部分表示一次最近的访问，浅色阴影表示一次较早的访问，白色意味着还没有被访问到。

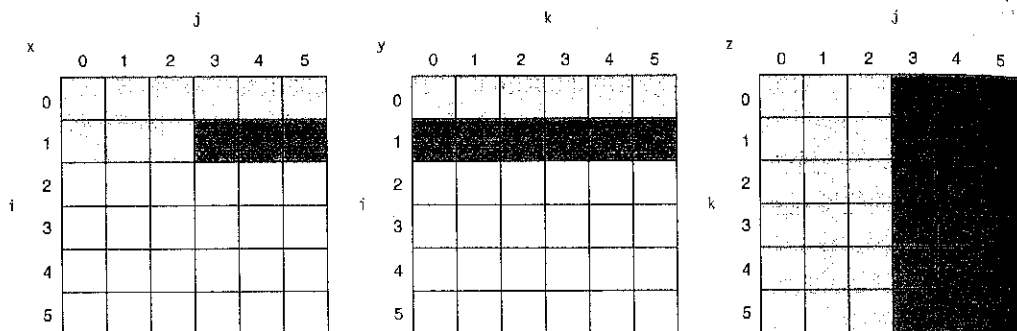


图 5.8 $N=6$ 和当 $i=1$ 时，数组 x 、 y 和 z 的示意图。访问数组元素的历史用阴影标出：白色表示还没有访问到，浅色表示较早的访问，而深色是最近的访问。变量 i 、 j 和 k 按照访问数组的行或列给出。相对图 5.9，在计算 x 的元素时， y 和 z 的元素是重复读取的

容量缺失显然依赖于 N 和 Cache 的容量。如果它能够包含所有 3 个 $N \times N$ 矩阵，则效果还会令人满意，但前提是不发生 Cache 冲突。如果 Cache 能包含一个 $N \times N$ 矩阵以及一行的 N 个元素，那么至少 y 的第 i 行以及数组 z 可以保留在 Cache 中。若 Cache 的容量更小，则对于 x 和 z 都会有缺失发生。最坏的情况下，进行 N^3 次操作，会有 $2N^3 + N^2$ 个字从存储器中被读出。

为了保证要访问的元素都能在 Cache 中命中，对源代码进行了修改，使其在一个 $B \times B$ 的子矩阵上计算。现在两个内部循环按步长 B 计算，而不是从 x 和 z 的开始一直到结束来计算， B 称为分块因子（假设 x 的初始值为 0）。

```

/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
    for (j = jj; j < min(jj+B,N); j = j+1)
        {r = 0;
        for (k = kk; k < min(kk+B,N); k = k + 1)
            r = r + y[i][k]*z[k][j];
        x[i][j] = x[i][j] + r;
        };

```

图 5.9 显示了用分块方法来访问 3 个数组的情况。仅仅观察容量缺失，总共访问的存储器字为 $2N^3/B + N^2$ ，大约改进了一个因子 B 。因此，分块充分利用了访问的空间局部性和时间局部性： y 是从空间局部性中得到改进，而 z 是从时间局部性中得到改进。

虽然我们的目标是降低 Cache 缺失，但是分块也有助于寄存器分配。通过采用一个小的分块，使得整个块能够被装入到寄存器中，可以使程序中 load 和 store 操作的数量最小化。

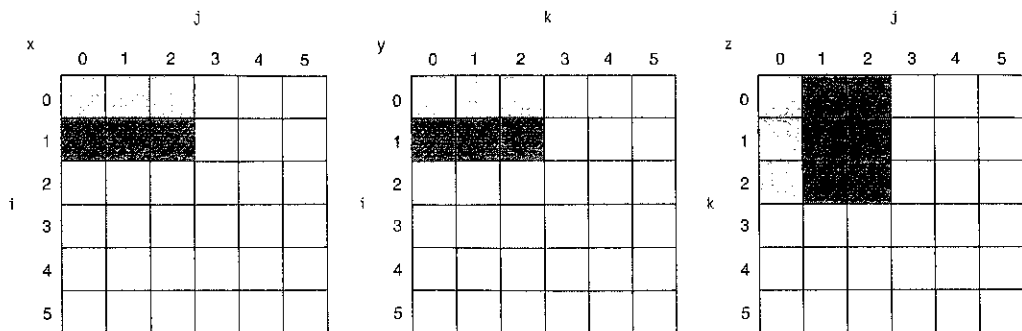


图 5.9 当 $B=3$ 时, 访问数组 x , y 和 z 的历史。注意同图 5.8 比较时, 被访问的元素较少

第十种优化: 指令和数据硬件预取以降低缺失代价/缺失率

非阻塞Cache通过重叠存储器访问和执行时间有效地降低了缺失代价。另外一种方法是在处理器访问指令和数据之前, 就把它们预取到Cache或预取到可以比存储器访问速度更快的外部缓冲区中。

指令预取通常在Cache之外的硬件中完成。一般情况下, 微处理器在缺失时取两个块: 被请求的块和其相邻的块。被请求的块装入到指令Cache中, 而预取的块则装入到指令流缓冲区中。如果被请求的块已经在指令流缓冲区中, 则取消原始的Cache请求, 从流缓冲区读入该块, 并发出下一个预取请求。

相似的方案可以应用到数据访问中[Jouppi 1990]。Palacharla和Kessler[1994]对一组科学计算程序以及能够处理指令或数据的流缓冲区进行了探索。研究发现对于带有两个容量为64 KB的4路组相联Cache的处理器, 其中一个是指令Cache, 另一个是数据Cache, 8个流缓冲区可以捕捉到50%~70%的缺失。

Intel Pentium 4 能从8个分别来自不同的4 KB页面的流中, 预取数据到二级Cache。如果二级Cache对同一个页面有两次连续的缺失, 并且这些Cache块的间隔小于256字节(Pentium 4中的某些模型的间隔门限是512字节), 就会调用预取。预取不会超越4 KB页面。

图5.10显示了当启用硬件预取时, SPEC2000的一个子程序测到的全局性能改进的情况。注意, 图中的12段程序中只有2段是定点程序, 而大部分都是SPEC浮点程序。

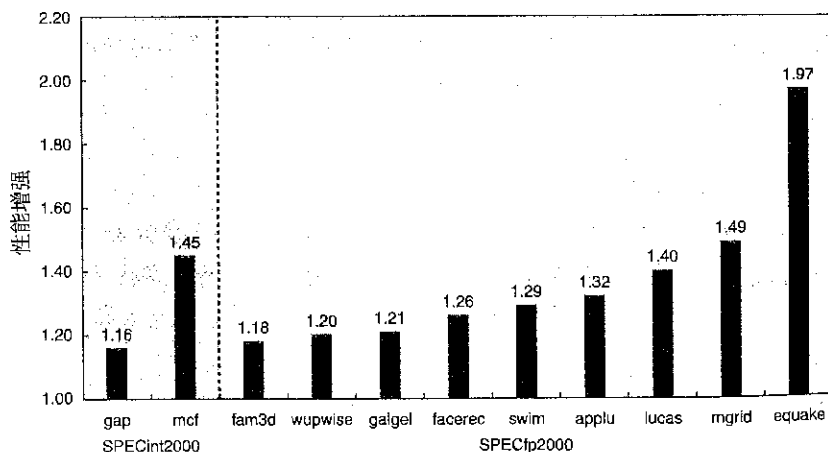


图 5.10 在 Intel Pentium 4 开启硬件预取, 加速了 12 个中的 2 个 SPECint2000 基准测试和 14 个中的 9 个 SPECfp2000 基准测试。图中仅仅列出了采用预取技术后性能改进最大的程序, 预取加速了 15 个 SPEC 基准测试程序, 减少了 15% 的缺失 [Singhal 2004]

预取技术会占用存储器带宽，但是它造成缺失增多，实际上会降低性能。借助于编译器，可以减少不必要的预取操作。

第十一种优化：编译控制预取降低缺失代价/缺失率

硬件预取的一个替代方案是利用编译器来插入预取指令，提前发出数据请求。有两种预取的方法：

- **寄存器预取**：把值预取到寄存器中。
- **Cache 预取**：只把数据预取到 Cache 中，并不放到寄存器中。

这两者都可以是故障性的或是非故障性的：区别在于地址会不会引起虚拟地址错误异常和保护冲突异常。如果用这个术语表述，一个正常的 load 指令可能就会称为是一个“故障性寄存器预取指令”。非故障性预取指令通常在导致异常发生时仅仅转化为空操作，这正是我们想得到的。

最有效的预取对程序而言是“语义透明”的：它不改变寄存器和存储器的内容，也不引起虚拟存储器错误。当今大多数处理器采用非故障性 Cache 预取技术，也称为非限定预取，本节也假定使用非故障 Cache 预取。

只有当预取数据的同时处理器能够继续工作，预取才有意义，也就是说，当等待预取数据返回的同时，Cache 并不停止，而是可以继续提供指令和数据。这种类型称为非阻塞 Cache。

正如硬件控制的预取一样，编译控制预取的目的是使执行和预取数据重叠进行。循环是主要对象，因为它们很适合于预取优化。如果缺失代价小，编译器只需要把循环展开一次或两次，然后在执行时进行预取调度。如果缺失代价较大，就要使用软件流水线（见附录 G）或是多次展开循环来为后续的循环预取数据。

然而生成预取指令会带来指令的开销，因此，我们必须谨慎使用它以保证这些开销在可接受的范围内。通过分析有可能发生 Cache 缺失的存储器访问操作，程序能够避免不必要的预取，同时显著地缩短存储器访问时间。

例题 对于下面的代码，首先确定哪些访问容易引起数据 Cache 缺失。然后插入预取指令来减少缺失。最后，计算被执行的预取指令数量以及由于预取而避免的缺失数量。假定数据 Cache 容量为 8 KB，块容量为 16 字节，采用直接映射，执行写分配的写回法。a 是 3 行 100 列的数组，b 是 101 行 3 列的数组，而且 a 和 b 中的每一个元素都是 8 字节长的双精度浮点数。我们还是假定程序初始时，数组中的数据不在 Cache 中。

```
for (i = 0; i < 3; i = i+1)
    for (j = 0; j < 100; j = j+1)
        a[i][j] = b[j][0] * b[j+1][0];
```

解答：编译器首先要确定哪些访问容易引起 Cache 缺失；否则会为本来可以命中的数据产生预取指令而浪费时间。数组 a 的元素按照它们在存储器中存储的顺序被写入，所以 a 会受益于空间局部性；当 j 为偶数值时将会缺失，为奇数值时则会命中。因为 a 有 3 行 100 列，它的访问将会产生 $3 \times \left\lceil \frac{100}{2} \right\rceil$ 即 150 次缺失。

数组 b 不能从空间局部性中受益，因为不是按照存储顺序来对它进行访问的，但它可以从时间局部性中受益两次：i 的每次循环会访问同样的元素，而 j 的每次循环都使用了与上次循环相同

的 b 的值。忽略潜在的冲突缺失, 当 $i=0$ 时会发生因访问 b 而引起的缺失, j 从 0 循环到 99, 所以对 b 的访问会产生 $100+1$ 即 101 次缺失。

这个循环将会引起数据 Cache 缺失 $150+101$ 次, 共有缺失 251 次。

为了简化优化策略, 将忽略循环中第一次访问的预取。这些值可能已经在 Cache 中, 不然将为取得数组 a 和 b 的前几个元素而付出缺失代价。我们也不考虑在循环末尾取消预取的情况, 即不考虑超出数组 a 的末尾 ($a[i][100] \cdots a[i][106]$) 和超出 b 的末尾 ($b[101][0] \cdots b[107][0]$) 的预取。如果它们是故障性预取, 我们不会采用这些开销很大的手段。假定缺失代价很大, 以至于需要至少提前 7 次循环进行预取 (换言之, 如果预取提前的循环次数小于 8, 就得不偿失了), 在程序中需要预取的部分已用下划线标出。

```
for (j = 0; j < 100; j = j+1) {
    prefetch(b[j+7][0]);
    /* b(j,0) for 7 iterations later */
    prefetch(a[0][j+7]);
    /* a(0,j) for 7 iterations later */
    a[0][j] = b[j][0] * b[j+1][0];
}
for (i = 1; i < 3; i = i+1)
    for (j = 0; j < 100; j = j+1) {
        prefetch(a[i][j+7]);
        /* a(i,j) for +7 iterations */
        a[i][j] = b[j][0] * b[j+1][0];
    }
```

这段修改过的代码预取到的元素是从 $a[i][7]$ 到 $a[i][99]$, 从 $b[7][0]$ 到 $b[100][0]$, 从而将没有预取措施的 251 次缺失减小到

- 第一次循环取 $b[0][0]$, $b[1][0]$, \cdots , $b[6][0]$ 的 7 次缺失。
- 第一次循环取 $a[0][0]$, $a[0][1]$, \cdots , $a[0][6]$ 的 $4 \times ([7/2])$ 次缺失 (空间局部性使每个 16 字节的块的缺失次数减小到 1 次)。
- 第二次循环取 $a[1][0]$, $a[1][1]$, \cdots , $a[1][6]$ 的 $4 \times ([7/2])$ 次缺失。
- 第二次循环取 $a[2][0]$, $a[2][1]$, \cdots , $a[2][6]$ 的 $4 \times ([7/2])$ 次缺失。

总共 19 次非预取性缺失。通过执行 400 次预取指令避免了 232 次 Cache 缺失, 这应该是很划算的。

例题 计算上例中节省的时间。忽略指令 Cache 缺失并且假定在数据 Cache 中不存在冲突缺失和容量缺失。假定预取操作可以彼此重叠也可以和 Cache 缺失操作重叠, 因此, 可以在最大存储器带宽下进行信息传输。下面给出不计 Cache 缺失的核心循环执行时间: 在原先的循环中, 每次迭代的执行要花费 7 个时钟周期, 原先的循环经过修改后, 第一个预取循环每次迭代的执行需要 9 个时钟周期, 而第二个预取循环每次迭代的执行需要 8 个时钟周期 (包括循环外部的开销)。一次缺失要用 100 个时钟周期。

解答: 原先的两层嵌套循环执行 $3 \times 100 = 300$ 次乘法。因为迭代的执行需要用 7 个时钟周期, 所以, 总循环时间为 $300 \times 7 = 2100$ 个时钟周期再加上 Cache 缺失时间, Cache 缺失时间为 $251 \times 100 = 25100$ 个时钟周期, 所以总的时间为 27200 个时钟周期。在对循环代码进行修改后, 第一个预取循环重复执行 100 次; 每次执行需要 9 个时钟周期, 总共是 900 个时钟周期再加上 Cache 缺失时间 $11 \times 100 = 1100$ 个时钟周期, 得到的时钟周期数是 2000。第二个预取循环重复执行 $2 \times 100 = 200$ 次, 每次执行需要 8 个时钟周期, 得到 1600 个时钟周期, 再加上

$8 \times 100 = 800$ 个 Cache 缺失的时钟周期，这样得到的时钟周期总数是 2400。从上例中我们已知这段代码在 $2000 + 2400 = 4400$ 个时钟周期内执行了 400 个预取指令，完成了上面的两个循环。如果假定预取操作与其余的执行部分可以完全重叠，那么这段预取代码要快 $27\ 200/4400$ 即 6.2 倍。

尽管数组优化易于掌握，但很多程序员更喜欢用指针。Luk 和 Mowry [1999] 指出：基于编译的预取有时也能扩展到指针上。在 10 个有递归数据结构的程序中，当一个节点被访问时，预取所有的指针能将半数程序的性能提高 4%~31%，在另外一半程序中，性能只提高 2%。预取的数据是否已在 Cache 中、预取指令的出现时间是否早于处理器用到该数据的时间，是两个必须要考虑的因素。

Cache 优化技术小结

改善命中时间、带宽、缺失代价及缺失率的多种技术通常会对平均存储器访问时间公式中的其他因子产生影响，同时也会影响存储器层次结构的复杂性。图 5.11 总结了这些技术，并对这些技术对存储器层次结构复杂性的影响进行了评价，使用“+”表示该技术改善相应特性，使用“-”表示该技术损害该特性，空白表示没有影响。一般而言，几乎没有一种技术能够改善多种性能。

技术	命中时间	带宽	缺失代价	缺失率	硬件成本/ 复杂度	备注
小而简单的 Cache	+			-	0	小，广泛使用
路预测 Cache	+				1	在 Pentium 4 中使用
踪迹 Cache	+				3	在 Pentium 4 中使用
流水线访问 Cache	-	+			1	广泛使用
非阻塞 Cache		+	+		3	广泛使用
多组 Cache		+			1	在 Opteron 和 Niagara 的二级 Cache 中使用
关键字优先与提前重启动			+		2	广泛使用
并写缓冲区			+		1	与写直达法一起广泛使用
采用编译技术				+	0	软件设计是关键，一些计算机提供编译器选项
指令和数据的硬件预取			+	+	指令为 2 数据为 3	大部分预取指令；Opteron 和 Pentium 4 预取数据
编译控制的预取			+	+	3	需要非阻塞 Cache；可能带来指令开销；大部分处理器支持

图 5.11 11 种 Cache 优化技术总结以及各种技术对 Cache 复杂度和性能的影响。通常一种技术只改善某一个性能指标。预取时机如果选择适当，则能降低缺失次数；否则，它也会降低缺失代价。“+”表示该技术改善相应特性，“-”表示该技术损害该特性，空白表示没有影响。复杂度的定义具有主观性，0 表示最简单，3 表示最困难

5.3 存储器技术及性能优化

……推动计算机迅速发展的一项重要成果就是可靠存储体的发明，也就是说，核心存储器……它不仅价格合理，而且性能可靠，因为可靠，所以我们可以必要时扩展它的规模。[p.209]

Maurice Wilkes
Memoirs of a Computer Pioneer (1985)

存储器是存储层次中 Cache 的下一级存储器。存储器满足 Cache 的访问需求并作为 I/O 接口,因为它既是输入的目的地,也是输出的源。存储器的主要性能指标是时延和带宽。一直以来,Cache 设计关注的是存储器时延(它直接影响 Cache 缺失代价),而 I/O 和多处理器设计所关注的是存储器的带宽。存储器和多处理器、I/O 之间的关系分别在第 4 章和第 6 章中描述。

从 Cache 角度来看,希望存储器越快越好。但是,降低存储器时延是很困难的,而通过采用新的组织形式来增加存储器带宽却相对容易些。随着二级 Cache 的广泛使用以及 Cache 中块容量的不断增加,存储器带宽对 Cache 也变得越来越重要。事实上,Cache 设计者可以通过增加块容量来利用存储器较高带宽的优势。

在前面几节中描述了可以通过在处理器和 DRAM 之间加入 Cache 以减少它们之间的性能差距,但是仅通过增大 Cache 或增加 Cache 级数并不能完全消除这样的性能差距。因此,存储器结构的创新也是必要的。

过去一段时间,对存储器的改进主要是考虑如何将更多 DRAM 芯片组织成存储器,例如多存储体技术。通过加宽存储器或总线,或者两者都加宽,为存储体技术带来了更高的可用带宽。

但是,随着每片存储芯片的容量增大,同样大小的存储器系统的芯片数量越来越少,这样反倒减少了创新的机会。例如,一块 2 GB 的存储器由 256 个 64 Mb ($16\text{M} \times 4\text{ bit}$) 存储芯片组成,它很容易组成 16 个 64 位宽的 16 路存储体芯片。但是,对于只需要 16 个 $256\text{M} \times 4\text{ bit}$ 的存储芯片的 2 GB 存储器,最多只能有一个 64 位宽的存储体。由于体积小且标准化的存储器配置是计算机的购买评价基准,厂商无法利用很大的存储器来提高带宽。在标准化配置中,芯片数量的大幅减少降低了板级改进的重要性。

因此,目前较多存储器创新是在 DRAM 芯片内部。这一节我们将介绍存储器芯片及其内部组织方法的技术改进。在此之前,我们先回顾一下存储器的性能参数。

存储器时延通常用两个参数来描述——存储器访问时间和存储周期。访问时间是指从发出读请求到返回被请求数据之间的时间,存储周期指连续两次存储器请求所需要的最小时间间隔。由于必须在地址线稳定后才能进行下一次存储器访问,所以存储周期要大于存储器访问时间。

事实上所有桌面计算机或服务器从 1975 年开始,就使用 DRAM 作为存储器,并使用 SRAM 作为 Cache,接下来首先介绍 SRAM。

SRAM 技术

SRAM 的第一个字母 S 代表静态。电路的动态特性要求 DRAM 在数据被读出之后需要写回操作——因此就产生了访问时间和存储周期的差异,以及必要的刷新。而 SRAM 不需要刷新,故这里访问时间非常接近存储周期。SRAM 的特色是每位使用 6 个晶体管来存储,以防止信息在读取时被破坏。SRAM 仅仅需要最小限度的电量来保持信息不丢失。

SRAM 在设计时尤其关注速度和容量,而 DRAM 则关注容量和每位的成本。对于同等工艺的存储器,DRAM 的容量大概是 SRAM 的 4~8 倍;SRAM 的存储周期比 DRAM 的快 8~16 倍,同时价格也比 DRAM 贵 8~16 倍。

DRAM 技术

在早期,随着 DRAM 容量的增大,地址线的数目也需要相应地增加,从而导致地址线的封装成本过高。一种解决方法是采用多路复用技术,使地址线减少一半。图 5.12 给出了 DRAM 的基本组织结构。一个地址分两部分传送,在行选通(RAS)方式下传送前半地址,而后在列选通(CAS)

方式下传送后一半地址。这种方法的原理是基于存储器内部组织结构的, 因为存储位是以矩阵形式组织的。

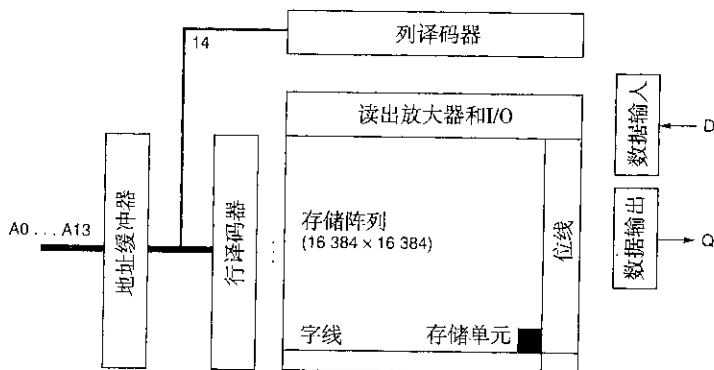


图 5.12 16 Mb DRAM 的内部结构。DRAM 在内部可以使用和选择不同的存储阵列。例如, 可以用 256 个 1024×1024 阵列或 16 个 2048×2048 阵列代替一个 16384×16384 阵列

对 DRAM 的另一个需求来自于它的第一个字母 D——动态特性。为了在一块芯片中存储尽可能多的信息, DRAM 只使用一个晶体管来存储一位信息。在读取这一位时会破坏该信息, 因此必须对信息进行恢复。这是 DRAM 的存储周期比访问时间要长的原因之一。另外, 为了防止每一位在未被读或未被写时信息不丢失, 也必须周期性地刷新每一位。幸运的是, 一行中的所有位在读取这一行时可以被同时刷新。因此, 存储系统中每一个 DRAM 必须在某一时间窗口内 (如 8 ms) 定期地访问每一行。在存储器的控制电路中有专门的硬件来周期性地刷新 DRAM。

DRAM 的这一特点意味着存储器系统在它发信号进行芯片刷新的瞬间是不可用的。刷新所需要的常规时间为访问全部 DRAM 内容 (包括 RAS 和 CAS) 所需时间的总和。由于 DRAM 中的存储器矩阵是一个平方的概念, 刷新所需的步数通常为 DRAM 容量的平方根。DRAM 设计者通常力求使存储器刷新时间小于存储器访问总时间的 5%。

前面的章节里把存储器的运作描述得好像一辆瑞士火车, 根据时间表始终如一地把货物准确地送到目的地。但是 DRAM 的刷新需求使这种理想的比喻难以实现, 因为某些访问操作的耗时比其他操作要长得多。所以, 刷新也是造成存储器执行时间不一致并导致 Cache 缺失代价变化的原因之一。

Amdahl 提出过一个著名的经验规律: 存储器容量应随着处理器速度的提高而线性增大, 从而保持系统平衡。根据该规则, 一个 1000 MIPS 的处理器应该配备 1000 MB 的存储器。设计者们希望借助于 DRAM 来实现这一需求, 即期望每三年存储容量增长 4 倍, 或每年增长 55%。遗憾的是, DRAM 性能增长的速度远低于这一速度。从图 5.13 可以看到, 和时延密切相关的行访问时间每年只提高 5%, 而和存储器带宽密切相关的列访问时间或数据传输时间每年提高的比率约为 10% 左右。

虽然我们讲述的是单独的 DRAM 芯片, 但实际上 DRAM 通常是被放置在称为双列直插存储模块 (DIMM) 的小电路板上出售的。DIMM 通常包含 4~16 个 DRAM 芯片。它们通常被组织成 8 字节宽, 适合于桌面系统使用。

本节除了讲述下面讨论的改善数据传输时间的 DIMM 封装和接口问题外, 还会提到 DRAM 容量增长速度缓慢这一最大的变化。在过去的 20 年内, DRAM 的容量已经按照摩尔定律以 3 年翻两番的速度增长。但是从 1998 年起, 由于对 DRAM 需求的减少, 其增长速度降为每两年翻一番。在 2006 年, 其增长速度依然没有加快的迹象。

年份	芯片大小	行选通(RAS)		列选通 (CAS)/ 数据 传输时间 (ns)	周期 时间 (ns)
		最慢的 DRAM (ns)	最快的 DRAM (ns)		
1980	64 Kb	180	150	75	250
1983	256 Kb	150	120	50	220
1986	1 Mb	120	100	25	190
1989	4 Mb	100	80	20	165
1992	16 Mb	80	60	15	120
1996	64 Mb	70	50	12	110
1998	128 Mb	70	50	10	100
2000	256 Mb	65	45	7	90
2002	512 Mb	60	40	5	80
2004	1 Gb	55	35	5	70
2006	2 Gb	50	30	2.5	60

图 5.13 每一代 DRAM 的最快和最慢访问时间 (存储周期在 215 页定义)。自 1986 年开始, 在从 NMOS DRAM 到 CMOS DRAM 的发展过程中, 行访问时间每年改进 5%, 列访问时间的改进速度是行访问时间的 2 倍

在 DRAM 芯片内部改善存储器性能

随着处理器的速度按照摩尔定律稳定增长, 处理器和存储器之间的性能差异越来越大, 因此对提高存储器性能的需求也日益紧迫。前面章节提到的一些思想也已经陆续应用到 DRAM 中, 但是它们基本都是靠增大存储器执行时延来换取高带宽的。下面将介绍一种利用 DRAM 自身特点的技术。

前面提到, DRAM 访问被分为行存取和列存取两步。所以在列选通之前, DRAM 必须缓冲一行中的所有位, 而一行中的位数通常是 DRAM 的大小的平方根, 即 256 Mb 的 DRAM 每行有 16 Kb, 1 Gb 的 DRAM 每行有 64 Kb, 依此类推。

尽管我们在逻辑上把 DRAM 表示成一个单片的存储位阵列, 但实际上其内部是由多个存储模块组成的。由于各种制造上的原因, 这些模块的容量通常在 1~4 Mb 之间。所以如果你在显微镜下观察一个 1 Gb 的 DRAM, 你可能会看到 512 个 2M 的存储阵列。由于阵列的数目庞大, 这就使实现更高的带宽成为可能。

一直以来, 有很多创新技术以提高带宽为目的。第一种是快速页模式, 它用同步信号在不需要额外行访问时间的前提下, 实现对行缓冲区的重复访问。这样的缓冲区易于实现, 每一个阵列会为每一次访问缓存 1024~2048 位。

传统的 DRAM 和存储器控制器之间是一个异步的接口, 所以与控制器之间的同步会增加每次数据传输的开销。第二种方法是在 DRAM 接口中增加一个时钟信号, 使得重复的传输不会增加这一开销, 这种优化方法称为同步 DRAM (SDRAM)。SDRAM 内部有一个可编程的寄存器, 保存了每次请求的字节数, 所以它可以为每个请求在几个时钟周期内传送大量的字节。

第三种增加带宽的主要改进是在 DRAM 的时钟脉冲的上升沿和下降沿都传送数据, 这样就可以倍增峰值速率。这称为双倍数据传输 (DDR)。为了实现以这样的高速率来提供数据, DDR DRAM 在内部激活了多存储体。

DRAM 的总线频率仍然是 133~200 MHz, 但是这些 DDR DIMM 经常以其峰值带宽标称。例如, DIMM PC2100 名称的由来就是: $133 \text{ MHz} \times 2 \times 8 \text{ bytes} = 2100 \text{ MB/s}$ 。为了避免这样的混淆,

芯片自身要使用每秒钟的位数而不是时钟频率来标识，因此 133 MHz 的 DDR 芯片称为 DDR266。图 5.14 给出了时钟频率、每个芯片每秒钟的传输率、芯片命名、DIMM 带宽和 DIMM 名称之间的关系。

标准	时钟频率 (MHz)	每秒钟 传送的 字节 (MB)	DRAM 名称	每分钟 DIMM 传送 的字节 (MB)	DIMM 名称
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10 664	PC10700
DDR3	800	1600	DDR3-1600	12 800	PC12800

图 5.14 2006 年 DDR DRAM 和 DIMM 的命名以及其时钟频率、带宽。请注意图中每列之间的数量关系。第三列是第二列的 2 倍，第四列使用第三列的数字来命名 DRAM 芯片。第五列是第三列的 8 倍，同时第五列的数字又用来命名 DIMM。虽然图中并未显示，但 DDR 还是会导致时延。以 DDR400 CL3 命名意味着存储器在开始传输数据时延迟 3 个时钟周期（每个 5 ns，因为时钟频率是 200 MHz）。下面的习题会研究更进一步的细节

例题 假设测量到新的 DDR3 DIMM 的传输率是 1600 MB/s。你认为这样的 DIMM 叫哪种名称比较合适？这种 DIMM 的时钟频率是多少？使用这样的 DIMM 的 DRAM 的名称如何？

解答：假定 DRAM 的市场人员认为使用最大的数字有助于销售，这样的 DIMM 名称应称为 PC16000，DIMM 的时钟频率是 1000 MHz，即每秒传输 2000 M，故 DRAM 的名称应称为 DDR3-2000。

$$\text{时钟比率} \times 2 \times 8 = 16\,000$$

$$\text{时钟比率} = 16\,000/16$$

$$\text{时钟比率} = 1000$$

DDR 现在已经成为一个标准序列，DDR2 从 2.5 V 降到 1.8 V，减少了耗电量，同时还提高了时钟频率，依次为 266 MHz，333 MHz 和 400 MHz。DDR3 把耗电降低到 1.5 V，并提供了 800 MHz 的最大时钟频率。

在这三个优化实例中，共同的优点是都增加了小部分的逻辑单元来利用 DRAM 内部潜在的高带宽，在系统额外开销很低的情况下，实现了带宽的重大改善。

5.4 保护：虚拟存储器和虚拟机

虚拟机是一个高效的、独立于真实计算机的复本。我们把这个概念解释为虚拟机监视器 (VMM)……VMM 有三个本质上的特性。首先，VMM 提供了一个程序运行的环境，它在本质上和原始计算机的环境相同；其次，在此环境中运行的程序在最坏的情况下也只有很少的速度损失；最后，VMM 能完全控制系统资源。

Gerald Popek 和 Robert Goldberg

“Formal requirements for virtualizable third generation architectures”

Communications of the ACM (July 1974)

2006年安全和保密是信息技术领域中最令人困扰的两个问题。人们时常会看到有关电子盗窃的报道,比如信用卡号泄漏。但是,那些未被公布出来的电子盗窃案可能更多。因此,不管是学术界还是业界都在探寻一种新的方法,使计算机系统更加安全。信息保护不仅仅局限于硬件中,在我们看来,真正的安全和保密很可能既包括计算机系统结构的创新,也涉及系统软件的改进。

本节首先会回顾计算机系统结构是如何通过虚拟存储器对不同的进程进行保护的。接着会描述另一种由虚拟机提供的保护机制,并介绍虚拟机系统结构的要求和虚拟机性能。

通过虚拟存储器来提供保护

页式虚拟存储器包含一个变换旁视缓冲器(TLB)或快表,其中的项和Cache页表项类似,它是保护各个进程互不干扰的主要机制。附录C中的C.4节和C.5节回顾了虚拟存储器,包括在x86上使用段式和页式方式保护的细节描述。这一部分只做简要的回顾,具体请参见附录。

随着多道程序设计出现,使得计算机被并行运行着的多个程序共享,这就要求在程序之间提供保护和共享机制,因而产生了进程的概念。进程可以比喻为一个程序呼吸的空气和存活空间,即一个运行着的程序连同继续运行它所必需的所有状态。在任何时候,必须能够从一个进程切换到另一个进程,这称为进程切换或上下文切换。

操作系统和系统结构结合在一起允许进程在共享硬件资源的同时又互不干扰。要实现这一目标,系统结构必须限制用户进程访问的权限,而增加操作系统进程的访问权限。系统结构设计至少需要完成以下任务:

1. 至少提供两种模式,指示当前运行的进程是用户进程还是系统进程,后者又称为**核心进程**或**管理进程**。
2. 提供一部分处理器状态信息,供用户进程使用但是不能写入。这包括一个用户/管理模式位、异常事件使能/禁止位和存储器保护信息。如果用户可以改变地址范围检查、拥有管理特权或能够禁止异常事件,那么操作系统将不能控制用户进程,因此,必须禁止用户进程对状态进行写操作。
3. 提供使处理器能够从用户模式切换到管理模式或反向切换的机制。第一种切换一般由**系统调用**完成,即通过一个特定的指令将控制转向管理代码空间中的一个特定位置。程序计数器在系统调用断点处被保存,处理器被置成管理模式状态。返回到用户模式类似于子程序返回,即恢复先前的用户/管理模式。
4. 提供限制存储器访问的机制,使得在进程切换时,能保存进程的存储器状态,而无须将进程与磁盘交换。

附录C介绍了几种存储器保护机制,目前应用最为广泛的是在虚拟存储器的每页中都加入保护约束条件。页面大小是固定的,其典型值是4 KB或8 KB,通过页表将这些页面从虚拟地址空间映射到物理地址空间。每页的保护约束条件包含在该页对应的页表项中。约束条件决定是否允许用户进程访问此页,代码能否从此页执行,还规定了如果进程不在此页,是否能读/写该页。由于只有操作系统能更新页表,故分页机制提供了完全的访问保护。

页式虚拟存储器意味着每次访问存储器时理论上需要两次操作,一次访存获得物理地址,另一次访存获得数据,这个代价自然太大了。一种补救的方法是利用局部性原理:如果访存具有局部性,

那么访存的地址转换也必然具有局部性。通过将这些地址转换保存在一个专门的 Cache 中,使得一次访存操作很少需要第二次访存去传输数据。这种特殊的地址变换 Cache 称为变换旁视缓冲器(TLB)或快表。

TLB 的一个项与 Cache 项相似,标志字段包含部分虚拟地址,数据部分包含物理页号、保护字段、有效位,通常还有一个使用位及一个重写位(dirty bit)。操作系统如果要改变这些位,它会首先改变页表中的值,然后通知 TLB 中相应的项无效。当这一项重新从页表中装载时,TLB 中的内容就和页表内容保持一致。

假设计算机完全遵守页约束条件和虚拟地址到物理地址的映射,那么这样看起来似乎就没有什么问题了,但频频发生的安全泄漏事件让我们认识到事情决不是这样简单的。

原因就在于我们不仅依赖于硬件的精确性,而且还依赖于操作系统的精确性。如今的操作系统由上千万行代码组成,按照惯例,我们以每一千行代码包含的 bug 数量来衡量软件可靠性,那么操作系统成品中包括数千个 bug 是很正常的。这些缺陷导致操作系统容易被攻击。

由于这一问题的存在,再加上现如今如果不采取保护措施将导致的高昂代价,使得人们去寻找这样一种保护模式,它比完整操作系统有更小的代码库,虚拟机就是其中的一个例子。

虚拟机的保护

虚拟机(VM)和虚拟存储器一样,都不是新的概念。虚拟机最早是在 20 世纪 60 年代末提出来的,这些年来它们都是大型机中的重要组成部分。虽然在 20 世纪 80 年代和 90 年代间,它们很少用于单用户计算领域,但最近受到人们的关注,这归结于以下原因:

- 在现代计算机系统中,隔离和安全性的重要性在增长。
- 标准操作系统在安全性和可靠性方面的缺陷。
- 在许多不相关的用户间共享单一的计算机。
- 处理器速度惊人的增长,使得虚拟机引起的开销降至可接受的范围内。

最广义的虚拟机定义包含所有基本的仿真方法来提供一个标准的软件接口,例如 Java 虚拟机。我们对虚拟机感兴趣的地方在于,在二进制指令集系统结构(ISA)的层次上提供一个完整的系统级环境。虽然一些虚拟机在本地硬件上运行不同的 ISA,但我们假设它都能匹配硬件。这样的虚拟机称为(操作)系统虚拟机,例如 IBM VM/370, VMware ESX Server 和 Xen。这些虚拟机让用户觉得自己在使用整台计算机,包括操作系统的副本。一台运行多个虚拟机的计算机可以支持多个不同的操作系统。在一个传统平台上,一个单独的“操作系统”拥有所有的硬件资源,但是通过使用虚拟机,多个操作系统可以共享硬件资源。

支持虚拟机的软件称为虚拟机监视器(VMM)或管理程序。VMM 是虚拟机技术的核心部分。在底层的硬件平台称为主机,它的资源在客户端虚拟机之间共享。VMM 决定了如何映射虚拟资源到实际的物理资源上,物理资源可能是通过分时共享、划分甚至是通过软件模拟的。VMM 比传统操作系统小很多,一个 VMM 的隔离部分也许只需要 10 000 行代码。

一般来说,处理器的虚拟化开销依赖于工作量。诸如 SPEC CPU2006 的用户级处理器边界程序没有虚拟化开销,因为操作系统在这里很少被调用,所以所有程序都能以原有速度来运行。I/O 强度密集的工作通常也是操作系统强度密集的,它们会执行许多系统调用和特权指令,因此导致很高的虚拟化开销。开销大小取决于需要由 VMM 进行模拟的指令数目,以及模拟速度的快慢。因此,当我们假设客户端虚拟机和主机运行同样的 ISA 时,系统结构和 VMM 的目标是尽可能将所有指令

在本地硬件上运行。另一方面,如果 I/O 强度大的负载同时也是 I/O 密集的,由于需要等待 I/O,处理器的虚拟化开销会完全被较低的处理器的利用率所掩盖(将会在稍后的图 5.15 和图 5.16 中看到)

尽管在这里我们对虚拟机提供的保护功能感兴趣,但虚拟机同时也提供了在商业上有重要意义的两个优势:

1. **软件管理。**虚拟机提供一个能运行完整软件堆栈的抽象,甚至包含像 DOS 这样的旧操作系统。虚拟机典型的部署包括:某些虚拟机运行旧的操作系统,大部分虚拟机运行流行的操作系统,少数的虚拟机用于测试操作系统的新版本。
2. **硬件管理。**需要多个服务器的一个原因,是为了让每个应用程序运行在一台单独的机器上,并拥有与之兼容的操作系统,这样的分隔能改善可靠性。虚拟机允许这些单独的软件栈能在共享硬件的同时独立运行,因而充当了多个服务器的角色。另一个例子是,一些 VMM 支持将正运行的虚拟机移植到另一台机器上,这样可以平衡负载或在硬件故障时实施迁移。

虚拟机监视器的必备条件

虚拟机监视器需要做什么?它提供给客户软件一个接口,分隔每个客户端的状态,并且需要把自己从客户端软件中隔离(包括客户操作系统)。定性的需求是

- 除了性能表现,和因多虚拟机共享而造成的固定资源限制以外,客户软件在虚拟机上的运行应该和它在本地硬件上的运行完全相同。
- 客户软件不能直接改变实际系统中的资源分配。

为虚拟化处理器, VMM 必须能控制一切——特权状态的访问、地址转换、I/O、异常和中断——这种控制即便是客户虚拟机和当前运行的操作系统临时正在使用它们时也完全不受影响。

例如,在计时器中断的情况下, VMM 需要挂起当前运行的客户虚拟机,保存其状态,处理中断,然后决定下面该运行哪个客户虚拟机,并加载其状态。依赖于计时器中断的客户虚拟机会由 VMM 为其提供的一个虚拟计时器和模拟的计时器中断。

客户虚拟机通常运行在用户模式下,为了方便管理, VMM 必须运行在一个更高的特权级别下。这也确保了执行任何特权指令都需要由 VMM 来处理。和上述页式虚拟存储器类似,系统级虚拟机的基本必备条件如下:

- 至少有两个处理器模式:系统级和用户级。
- 特权级指令只能在系统模式下使用,如果在用户模式下执行会产生 trap 中断。所有系统资源只能由这些指令控制。

虚拟机(缺乏)的指令集系统结构支持

如果在 ISA 设计过程中考虑到了虚拟机的使用,那么会相对容易地减少由 VMM 执行的指令数目和模拟这些指令所花费的时间。允许虚拟机直接在硬件上执行的系统结构被冠以可虚拟化的名称, IBM 370 就是如此。

由于虚拟机只是近期才在桌面系统和基于 PC 的服务器应用上考虑,故现有的大部分指令系统均没有考虑虚拟化的思想。80x86 和大部分 RISC 系统结构都是如此。

因为 VMM 必须确保客户系统只能和虚拟资源交互,故常规的客户操作系统在 VMM 的顶层运行用户模式程序。如果客户操作系统试图通过特权指令访问或修改相关硬件资源的信息,例如读写一个页表指针,那么它会向 VMM 发出 trap 中断。VMM 会做适当的调整以协调实际资源。

因此,如果任何指令试图在用户模式下读写这样敏感的信息 trap,那么 VMM 能截获 trap,并提供给虚拟机这些客户操作系统需要的敏感信息。

如果上述条件不具备,则需要其他方法。VMM 必须使用特别的预防措施来定位所有可能存在问题的指令,以确保它们能被客户操作系统正常执行,这就增加了 VMM 的复杂度并降低了虚拟机的运行性能。

5.5 节和 5.7 节给出了具体的在 80x86 系统结构下可能存在问题指令的例子。

虚拟机在虚拟存储器和 I/O 上的冲突

运行在每个虚拟机上的客户操作系统都管理着自己的页表,因此,另一个重要问题是虚拟存储器的虚拟化。为此,VMM 把实际存储器和物理存储器的概念(原本是同义的)区分开来,单独把实际存储器作为物理存储器和虚拟存储器中间的一层(也用虚拟存储器、物理存储器和机器存储器的概念来命名这三层)。客户操作系统使用页表将虚拟存储器映射到实际存储器,VMM 的页表又将用户的实际存储器映射到物理存储器。这样的虚拟存储器系统结构要么是由页表确定的,如 IBM VM/370 和 80x86,要么是由 TLB 结构确定的,如大部分 RISC 系统结构。

VMM 维持了一个影子页表来直接将客户虚拟地址空间映射到硬件物理地址空间,而不需要每次间接访问存储器而产生额外的开销。通过检查对客户页表的所有修改,VMM 就能确保硬件用来做地址转换的影子页表项和客户操作系统环境中页表项之间的一一对应关系,有一个例外是正确的物理页面替换了客户页表中的实际页。因此,VMM 必须 trap 中断任何试图通过客户操作系统来改变页表或访问页表的指针。这通常由写保护客户页表来实现,并且由客户操作系统 trap 中断任何访问页表指针的操作。如前所述,如果访问页表指针属于特权操作,那么后来发生 trap 中断是自然的。

IBM 370 系统结构在 20 世纪 70 年代解决了页表问题,它是通过由 VMM 管理一个附加间接访问层来实现的。客户操作系统和往常一样保持其页表,故不再需要影子页表。AMD 对 80x86 的改进版本 Pacifica,采用了与此类似的策略。

为了在大部分 RISC 计算机上虚拟化出 TLB 系统结构,VMM 管理实际的 TLB 并拥有每个客户虚拟机的 TLB 内容的副本。要实现这一功能,任何访问 TLB 的指令都需要 trap 中断。TLB 加上进程 ID 标签能支持不同虚拟机和 VMM 的混合项,这样可以避免在虚拟机切换时 TLB 被清除。与此同时,在后台 VMM 支持虚拟机的虚拟进程 ID 和实际进程 ID 间的映射。

系统结构中最后要虚拟化的部分是 I/O。由于和计算机相关的 I/O 数量和种类日益增多,到目前为止,它是系统中最难虚拟化的部分。另一个难点是一个实际设备在多个虚拟机之间的共享。此外,还有支持多种设备驱动的需求的问题,特别是在同一个虚拟机系统上支持不同的客户操作系统的情况。我们可以这样理解虚拟机:它为每种类型设备提供一个适用于每种客户虚拟机的通用驱动,并将管理实际 I/O 的职能留给 VMM。

映射虚拟 I/O 设备到物理 I/O 设备的方法依赖于设备类型。例如,物理磁盘通常由 VMM 划分给用户虚拟机创建虚拟磁盘,VMM 将虚拟磁道和扇区映射到物理磁盘上。网络接口通常是在很短的时间片内在虚拟机之间共享,VMM 的职能是对虚拟网络地址的消息进行跟踪,确保虚拟机只能收到发给自己的消息。

虚拟机监视器的实例: Xen 虚拟机

在虚拟机的早期发展中,最显著的问题是效率低下。例如,客户操作系统管理虚拟页面到实际页面的映射,而 VMM 会忽略此操作,并执行到物理页面的映射。换言之,客户操作系统浪费了大

量开销。为了提高效率, VMM 的开发者认为让客户操作系统感知其在虚拟机上的运行也许是个有效方法。例如, 客户操作系统能假定虚拟存储器和实际存储器大小一致, 这样就不需要客户操作系统来进行存储器管理。

允许对客户操作系统做细微的修改来简化虚拟化的方法, 称为泛虚拟化, 例如开源的虚拟机监视器 Xen。Xen 提供给客户操作系统一个和物理硬件相似的抽象虚拟机, 但它去掉了很多不利因素。例如, 为避免 TLB 被清除, Xen 将其自身映射到每个虚拟机前面的 64 MB 地址空间。它允许客户操作系统分配页面, 同时检查以确保其不会破坏保护约束。为了在虚拟机中保护客户操作系统不被用户应用程序破坏, Xen 利用了 80x86 中可用的 4 级保护机制。Xen VMM 运行在最高的特权 0 级, 客户操作系统运行在 1 级, 应用程序运行在最低的 3 级。大部分 80x86 的操作系统永远工作在特权 0 级或 3 级。

为使各部分协同工作, Xen 修改了客户操作系统, 使之不能使用可能引起问题的系统结构部分。例如, 对于 Linux, Xen 修改其端口上的 3000 行代码, 大约是使用在 80x86 上的代码的 1%。当然这些改动不会影响到客户操作系统提供给应用程序的二进制接口。

为简化虚拟机中 I/O 虚拟化的实现, Xen 将每个硬件 I/O 设备指派给特权虚拟机。这些特殊的虚拟机称为驱动域 (Xen 把其虚拟机称为域), 驱动域负责运行物理设备驱动, 不过在访问适当的物理设备之前, 仍然由 VMM 处理中断。普通的虚拟机, 也叫用户域, 运行着简单的虚拟驱动程序, 它必须和物理驱动在驱动域中通过一条专门的通道来通信, 进而访问物理 I/O 硬件。数据通过页面重映射, 并在用户域和驱动域之间传输。

图 5.15 通过 6 个基准测试程序比较了 Xen 的相关性能。实验表明, Xen 非常接近原始 Linux 的性能。由于 Xen 的广泛使用以及优异的性能表现, 在标准的 Linux 的新版本中也提供了对 Xen 的支持。

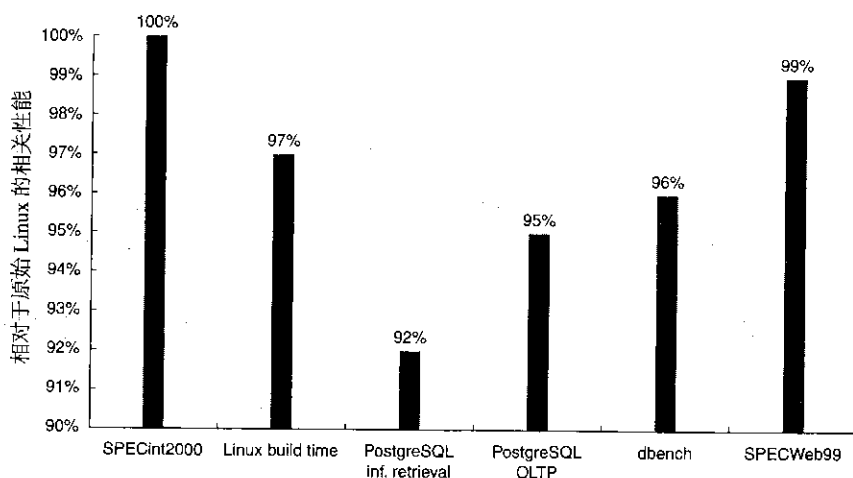


图 5.15 相对于原始 Linux, Xen 的相关性能。实验在 Dell 2650 上进行, 拥有双核 2.4 GHz Xeon 处理器, 2 GB 存储器, 1 块 Broadcom Tigon 3 Gigabit 以太网卡, 一块 Hitachi DK32EJ 的 146 GB 10 000 转的 SCSI 硬盘, 机器运行 Linux 2.4.21 版本内核 [Barham et al. 2003; Clark et al. 2004]

对图 5.15 的进一步研究会发现该实验是基于一块单独的以太网卡 (NIC) 的, 而这块单一的 NIC 是一个性能瓶颈。结果是, 虽然 Xen 提高了处理器利用率, 但性能并没有发生改变。图 5.16 在

原始 Linux 和两种 Xen 配置过的 Linux 上比较了 1 到 4 块网卡情况下 TCP 的接收性能。两种 Xen 的配置如下:

1. 仅 Xen 特权虚拟机 (驱动域)。为测量 Xen 在没有驱动虚拟机策略下的开销, 整个程序都在单一的特权域中运行。
2. Xen 客户虚拟机加上特权虚拟机。应用程序和虚拟设备驱动在客户虚拟机上运行, 而物理设备驱动在特权驱动虚拟机上运行, 这是更接近实际情况的配置。

很明显, 单一的 NIC 是性能瓶颈。Xen 驱动虚拟机在 2 块 NIC 情况下达到最大值 1.9 Gb/s, 原始 Linux 在 3 块 NIC 情况下达到最大值 2.5 Gb/s。对客户虚拟机来说, 峰值接收速率降到 0.9 Gb/s 以下。

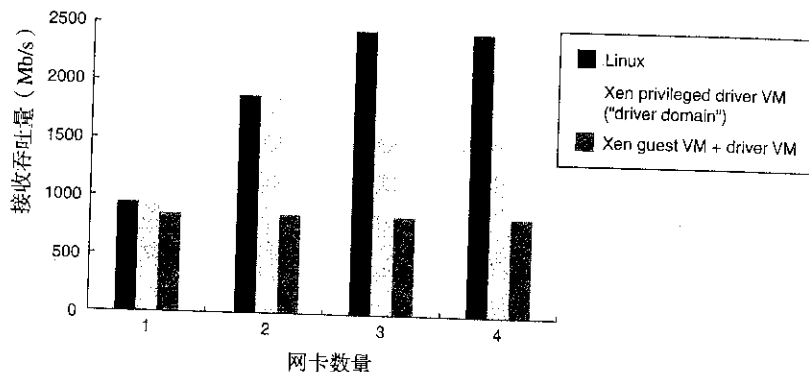


图 5.16 相对于 Xen 两种配置的原始 Linux 的 TCP 接收性能 (Mb/s)。客户虚拟机和驱动虚拟机都是常规的配置[Menon et al. 2005]。实验在 Dell PowerEdge 1600SC 上进行, 拥有 2.4 GHz Xeon 处理器, 1 GB 存储器, 4 块 Intel Pro-1000 Gigabit 以太网卡, 机器运行 Linux 2.6.10 版本内核和 Xen 2.0.3 版

在消除 NIC 瓶颈后, 不同的 Web 服务器负载显示 Xen 的驱动虚拟机在吞吐量方面是原始 Linux 的 80%, 而 Xen 客户虚拟机和特权虚拟机结合时, 才 34%。

图 5.17 给出了在 Xen 的两种配置和原始 Linux 环境中, 指令执行、二级 Cache 缺失、指令和数据 TLB 缺失的相对变化情况, 并据此分析了性能下降的原因。Xen 每条指令的数据 TLB 缺失比原始 Linux 高出 12~24 倍, 这是特权驱动虚拟机配置性能下降的主要原因。而较高的 TLB 缺失的原因, 是由于 Linux 使用了 Xen 所不具备的优化措施, 这包括: 超页面和全局标记页表的项。Linux 使用超页面作为其内核空间的一部分, 使用 4 MB 的页面显然比使用 4 KB 的页面降低了 TLB 的缺失。此外, 具有全局标记的 PTE 在上下文交换时不会被清除, Linux 在其内核空间正是使用了这些 PTE。

除了很高的 D-TLB 缺失之外, 更普遍使用的客户虚拟机加驱动虚拟机这种配置使指令执行的数量多出了两倍。这是由于页面重映射、驱动虚拟机和客户虚拟机的页面转换以及两个虚拟机在同一通道上的通信的额外开销造成的。这也是图 5.16 中指出的客户虚拟机接收性能较低的原因。另外, 客户虚拟机配置二级 Cache 缺失数多出 4 倍。原因是 Linux 使用零拷贝网络接口, 它依赖于 NIC 到存储器中不同位置进行 DMA 操作的能力。因为 Xen 不支持在其虚拟网络接口中进行聚集 DMA, 故它不能在客户虚拟机中进行真正的零拷贝, 这样导致了更多的二级 Cache 缺失。

尽管 Xen 的后续版本可能会支持超页面、全局标记 PTE 和聚集 DMA, 但客户虚拟机和驱动虚拟机的分离所带来的高指令开销似乎还是无法消除的。

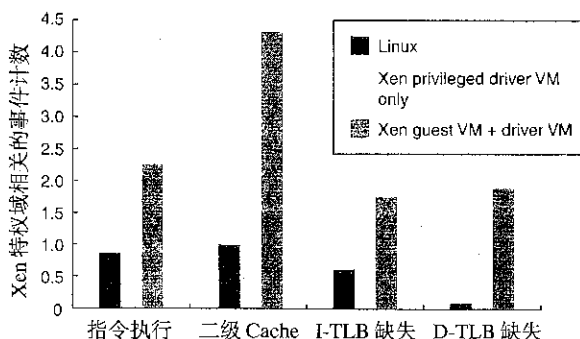


图 5.17 在 Web 负载情况下，原始 Linux 相对于 Xen 的两种配置在指令执行、二级 Cache 缺失、I-TLB 缺失和 D-TLB 缺失方面的相关变化[Menon et al. 2005]。较高的二级 Cache 缺失和 TLB 缺失是由于 Xen 不支持超页面、全局标记 PTE 和聚集 DMA[Menon 2006]

5.5 相关问题：存储器层次设计

这一节讲述其他章节中讨论过的、对存储器层次设计很重要的三个主题。

保护和指令集系统结构

强化保护措施需要同时依赖于操作系统和系统结构。但是，当虚拟存储器被广泛应用时，系统结构设计者就需要对指令集系统结构中一些很不方便使用的细节内容进行修改。例如，为了在 IBM 370 中支持虚拟存储器，系统结构设计者不得不改变仅仅使用了 6 年的 IBM 360 指令集系统结构。如今为了适应虚拟，也需要做相似的调整。

例如，80x86 的指令 POPF 从存储器堆栈的顶部加载标志寄存器。其中有一个标志是中断使能标志位 IE。如果在用户模式下运行 POPF 指令，它只是简单地改变除了 IE 位以外的所有标志位，而不是发生 trap 中断；但是在系统模式下，确实会改变 IE 位。这样就产生了一个问题，运行在虚拟机用户模式下的客户操作系统希望看到 IE 位的改变，但事实上它永远也看不到。

过去 IBM 的大型机硬件和 VMM 用以下三步来改善虚拟机的性能：

1. 降低处理器的虚拟化开销。
2. 降低由于虚拟化引起的中断开销。
3. 当中断发生时，直接转交给相应的 VM，而不用调用 VMM，从而降低了中断开销。

IBM 仍然掌握着虚拟机的核心技术。例如，IBM 的大型机上在 2000 年可运行数千个虚拟机，而 Xen 在 2004 年才运行了 25 个虚拟机[Clark et al. 2004]。

2006 年，AMD 和 Intel 提出新的计划致力于解决上述第一个方面的问题，即降低处理器的虚拟化开销（见 5.7 节）。然而，系统结构和 VMM 需要经过多少代的改进才能完全解决以上三方面的问题呢？需要经过多长时间，21 世纪的虚拟机能像 20 世纪 70 年代的 IBM 大型机和 VMM 一样有效呢？这些都是有趣的研究课题。

预测执行和存储系统

支持预测执行或条件指令的处理器可能产生无效地址，而不支持预测执行的处理器将不会产生这种情况。如果采用保护异常，这样不仅是不正确的表现，而且预测执行带来的益处还会被虚

假异常的开销所抵消。因此,存储系统必须识别预测执行指令和条件执行指令,并能抑制相应的异常。

同理,我们也不允许这样的指令引起Cache缺失而停顿,因为不必要的停顿可能抵消预测带来的优化效果,因此这些处理器必须与非阻塞Cache配合使用。

在实际应用中,二级Cache缺失的代价非常大,所以编译器一般只在一级Cache缺失时进行预测。从图5.5可知,对于一些优秀的科学计算程序来说,编译器可接受多次二级Cache缺失以大大减少二级Cache缺失带来的开销。另外,要使这些确实可行,还需要存储系统能够满足编译器并发访问存储器的需求。

I/O 和 Cache 数据的一致性

数据既可放在存储器中也可能放在Cache中。只要单处理器是改变和读取数据的唯一设备,并且Cache位于处理器和存储器之间,就或多或少地存在处理器访问到旧的或不一致的副本的风险。就像第4章提到的一样,多处理器和I/O设备增加了副本不一致或读到错误副本的概率。

对于多处理器来说,Cache一致性发生问题的频率和I/O发生问题的频率是不一样的。I/O很少发生多数据副本事件——无论何时,这都要尽可能地避免,但运行在多处理器上的程序需要在若干个Cache中有同样的数据副本。多处理器程序的性能依赖于共享数据时的系统性能。

I/O Cache一致性问题可表述为:I/O操作在计算机中的什么位置发生——是在I/O设备和Cache之间还是在I/O设备和存储器之间?如果输入是直接把数据输入到Cache中,而输出是从Cache中读出数据,那么I/O和处理器会看到相同的数据。这种方式的缺点是对处理器产生了干扰,会导致处理器因I/O而停顿。在向Cache输入数据时,要用一些新的数据替换旧的信息,由于这些数据可能不会被处理器很快访问,因此对Cache也产生了干扰。

在带有Cache的计算机中,I/O系统要尽可能少地占用处理器,同时要防止数据过期问题。因此,很多系统宁愿I/O操作直接在存储器中发生,把存储器作为I/O缓冲区。如果采用写直达Cache,那么,存储器总有一个最新的信息副本,在输出时也就不存在数据过期问题(这就是为什么很多机器采用写直达的原因之一)。但是,就目前来说,写直达法只用于一级Cache,而二级Cache则使用写回法。

输入要求做些额外的工作。软件的解决办法是保证输入缓冲区中的块都不在Cache中。一种方法是将一个包含这样的缓冲区的页标记成不可高速缓存(noncachable),操作系统总是向这页输入。另一种方法是操作系统在输入发生前从Cache中刷新缓冲区地址。硬件的解决方法是在输入时检查I/O地址,看它们是否在Cache中。如果在Cache中有一个与I/O地址相匹配的地址,则Cache项被置为无效以避免获取过期数据。所有这些方法也可用于写回Cache的输出。

5.6 综合: AMD Opteron 存储器层次结构

这一节将给出AMD Opteron存储器层次结构并列出现SPEC2000测试出的各组件性能。Opteron是一个乱序执行处理器,每时钟周期最多可以取3条80x86指令,然后将其转换成类RISC操作,它每时钟周期发射3条指令,而且有11个并行的执行单元。在2006年,12级定点流水线产生了2.8 GHz的最大时钟频率,支持的存储器最快达到PC 3200 DDR SDRAM,它使用的是48位虚拟地址和40位物理地址。图5.18指出了通过多级数据Cache和TLB的地址映射,这和图5.3中的类似。

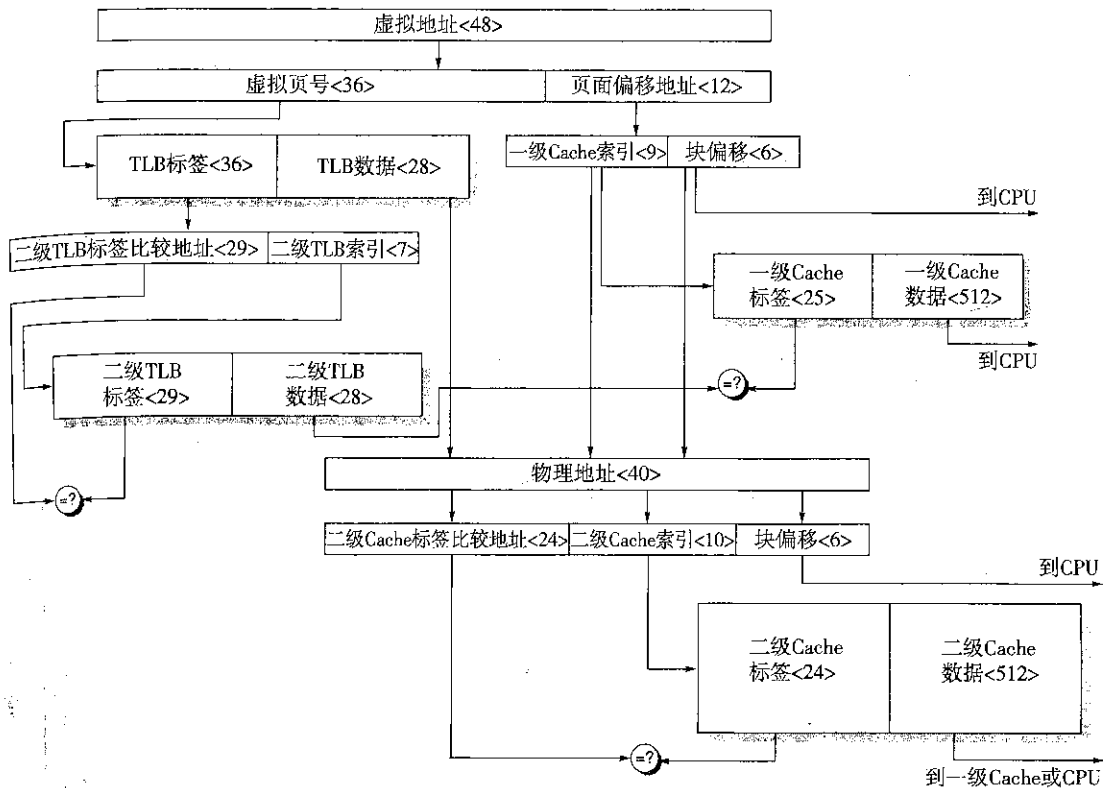


图 5.18 AMD Opteron Cache 和 TLB 的虚拟地址、物理地址、索引、标签以及数据分块。由于指令和数据的层次结构是对称的，因此我们仅描述一种。一级 TLB 全相联映射 40 个条目。二级 TLB 是 4 路组相联，映射 512 个条目。2 路组相联映射的一级 Cache 是 64 KB，16 路组相联映射的二级 Cache 是 1 MB，都是用 64 字节的块。此图未指出 Cache 和 TLB 的有效位和保护位，这会在图 5.19 中指出

我们现在开始逐步研究存储器层次结构的工作情况：图 5.19 以分步标注的形式描述了这一过程。首先，将程序计数器 PC 装入指令 Cache。Cache 的大小是 64 KB，采用 2 路组相联、64 字节的块，采用 LRU 替换策略。Cache 的索引长度为

$$2^{\text{Index}} = \frac{\text{Cache 大小}}{\text{块大小} \times \text{组相联度}} = \frac{64\text{K}}{64 \times 2} = 512 = 2^9$$

9 位索引。Cache 可以被虚拟地址索引和物理地址确认。首先，指令的数据地址的页面部分被送入 TLB 中（第 1 步），与此同时，虚拟地址的 9 位索引（加上额外的 2 位以便取出正确的 16 字节块）被送入数据 Cache（第 2 步）。全相联映射的 TLB 同时对全部 40 个条目进行搜索以发现地址与数据之间的有效匹配（第 3 步、第 4 步）。除了对地址进行转换外，TLB 还会检查这个 PTE 要求的数据是否会导致异常。

一个指令 TLB 缺失首先会到二级指令 TLB 中，它是一个 4 路组相联的 4 KB 页面，有 512 个条目。需要 2 个时钟周期将二级 TLB 加载到一级 TLB。在传统的 80x86 的 TLB 策略中，如果页面寄存器发生了变化，那么所有的 TLB 会被刷新。而 Opteron 则会检测存储器中发生变化的页面目录，仅仅刷新改变了的数据结构，这样就避免了不必要的刷新。

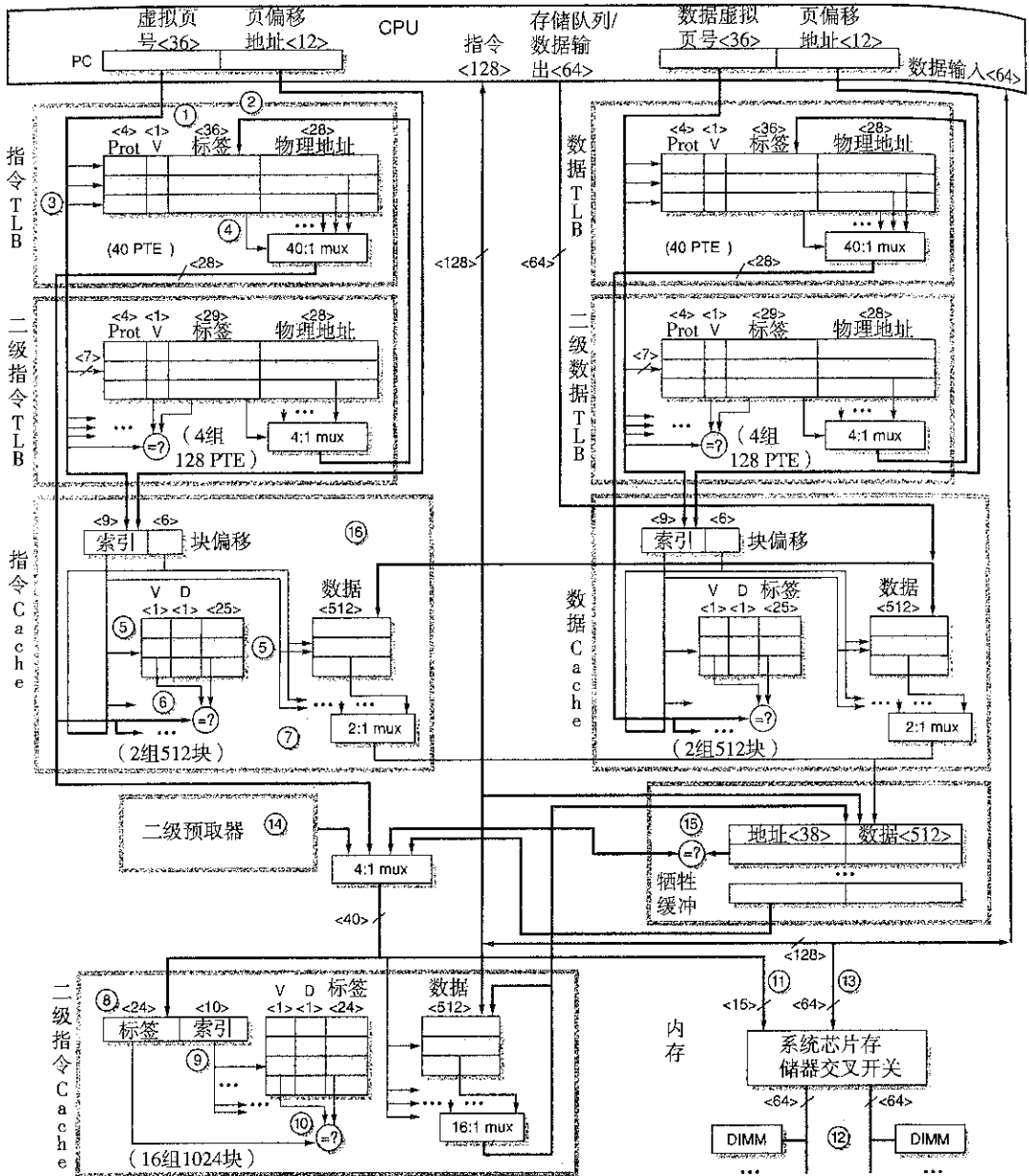


图 5.19 AMD Opteron 存储层次结构的完整图示。一级 Cache 是 64 KB 大小，2 路组相联，分块大小为 64 字节，采用 LRU 替换策略。二级 Cache 是 1 MB 大小，16 路组相联，分块大小也为 64 字节，采用伪 LRU 替换策略。写分配时，数据和二级 Cache 使用写回法。一级指令和数据 Cache 被虚拟索引并物理标注，因此在将每个地址发往 Cache 的同时，也必须送往指令或数据 TLB。这些 TLB 全部相联并有 40 个条目，其中 32 个条目是 4 KB 的页面，8 个条目是 2 MB 或 4 MB 的页面。在每个 TLB 之后，有一个 4 路组相联映射到 512 个条目，且每个页面大小为 4 KB 的二级 TLB。Opteron 支持 48 位虚拟地址和 40 位物理地址

最坏的情况是页面不在存储器中,操作系统需要从硬盘中将其调入。因为在页面错误时可以有上百万条指令在执行,所以,如果有另一个进程在等待执行,操作系统会将其调入并执行。另外,如果没有 TLB 异常发生,会继续访问指令 Cache。

地址域的索引同时被发送到了 2 路组相联数据 Cache 组中(第 5 步)。指令 Cache 的标记是 $40 - 9$ (实际索引长度) $- 6$ (块偏移) $= 25$ 位。将 4 位标签和有效位与指令 TLB 中的物理页面相比较(第 6 步)。因为 Opteron 期望每条指令取 16 字节,所以在 6 位块偏移外,还需要额外 2 位以便取出正确的 16 字节。因此,将 16 字节的指令发送给处理器需要 $9 + 2 = 11$ 位。一级 Cache 采用流水线技术,命中时间是 2 个时钟周期。缺失会同时转移到二级 Cache 和存储控制器上,这样在二级 Cache 缺失的情况下,可以降低缺失代价。

如前所述,指令 Cache 是虚拟地址寻址和物理地址标注。发生缺失时,Cache 控制器必须进行同义性检查(两个不同的虚拟地址引用同一物理地址)。所以,在查找二级 Cache 时,要把指令 Cache 标注检查和二级 Cache 标注检查并行进行。因为最小的页为 4 KB,需要 12 位,而 Cache 索引和块偏移共 15 位,所以 Cache 必须为每个同义引用进行块检查,即 23 个或 8 个块。Opteron 使用冗余侦听标记在一个时钟周期内检查所有同义性。如果找到了相同的引用,出错的块将被设置为无效。这样可以保证一个 Cache 块在任何时间都只会驻留在 16 个缓存区的某个中。

二级 Cache 试图在缺失时取出块。二级 Cache 是 1 MB 大小,16 路组相联,分块大小为 64 字节。它通过管理 8 对 LRU 块,使用伪 LRU 替换策略,然后随机选取一个 LRU 对来替换。二级 Cache 索引长度为

$$2^{\text{Index}} = \frac{\text{Cache 大小}}{\text{块大小} \times \text{组相联度}} = \frac{1024\text{K}}{64 \times 16} = 1024 = 2^{10}$$

所以 34 位块地址(40 位物理地址 $- 6$ 位块偏移)被分成 24 位标注和 10 位索引(第 8 步)。索引和标注再一次被送到所有 16 组 16 路组相联的数据 Cache 中(第 9 步),做并行的比较。如果有一个匹配且有效(第 10 步),它以每个时钟周期 8 字节的速度顺序返回该块。二级 Cache 也会取消一级 Cache 送入控制器中的存储器请求。如果一级指令 Cache 缺失在二级 Cache 中命中,在它的第一个字上将花费 7 个处理器时钟周期。

Opteron 在一级 Cache 和二级 Cache 之间使用一种独占策略来更好地利用资源,这意味着块要么在一级 Cache 中,要么在二级 Cache 中,不会同时在两者中。因此,它不是简单地将一个块的副本放入二级 Cache 中,而是仅仅将新块放入一级 Cache 中。一级 Cache 中过时的块被送入二级 Cache。如果二级 Cache 中的块是脏的,就会被送到写缓冲区,在 Opteron 中称为牺牲缓冲区。

在最后一章,我们讨论了相容性是如何允许所有的一致性流量仅影响二级 Cache 而不影响一级 Cache 的。而排斥性意味着一致性流量必须对上述两者进行检查。为降低一致性流量和一级 Cache 处理器之间的相互干扰,Opteron 为一致性侦听的地址标签提供了一组副本。

如果指令在二级 Cache 中没有找到,则片上的存储器控制器必须从存储器中获取块。Opteron 有 64 位双通道存储器,由于只有一个存储器控制器,两条通道上的地址相同,故可以作为一个 128 位的通道使用(第 11 步)。当两条通道有同样的 DIMM 时,可以有更宽的传输信道。每条通道最大支持 4 个 DDR DIMM(第 12 步)。

因为 Opteron 对数据 Cache、二级 Cache、总线和存储器提供单错误修正/双错误探测的检查,所以实际上数据总线每 64 位的数据里包括附加的 8 位 ECC 纠错码。为减少第二次出错的概率,Opteron 在空闲周期,通过读和重写数据 Cache、二级 Cache 和存储器上损坏了的块来解决单位错误。因为指令 Cache 和 TLB 是只读的数据结构,故它们由奇偶校验保护,如果发生奇偶错误,则从低级存储器重读。

加上关键指令的DRAM时延,由存储器提供服务的指令缺失的总共时延大约是20个处理器时钟周期。对于一个PC3200 DDR的SDRAM和2.8 GHz的CPU,对首个16字节,DRAM的时延是140个处理器周期(即50 ns)。存储控制器以每存储周期16字节的速率填满64字节大小的Cache块的剩余部分。对于200 MHz的DDR DRAM,多出3个时钟周期和7.5 ns的时延,即对于2.8 GHz的CPU会多出21个时钟周期(第13步)。

Opteron有一个和二级Cache相结合的预取引擎(第14步)。它控制二级Cache对连续块发生缺失时的处理策略,不论是向前连续还是向后连续,都把下一行预取到二级Cache中。

因为二级Cache采用写回法,所以任何一次缺失都会导致一个旧块被写回到存储器中。与数据Cache中脏的牺牲块的处理方式相同,Opteron会把这样的“牺牲”块放入牺牲缓冲区中(第15步)。缓冲区允许先执行原先缺失的指令读操作。Opteron把牺牲块的地址加在新请求的地址之后,通过系统地址总线送出,然后系统芯片取出牺牲块再写入DIMM。

牺牲缓冲区大小为8,所以在牺牲块被写到二级Cache或存储器中之前,缓存有足够容量来保存队列。存储控制器能最多并发处理10个Cache块缺失——包括8个数据Cache缺失和2个指令Cache缺失。而且允许它在10次缺失以下命中,详情请参见附录C中的描述。数据Cache和二级Cache检查牺牲缓冲区中的缺失块,但它仍然等待数据写到存储器中后,才重新去取。新的数据一旦到达,就立刻被载入指令Cache中(第16步)。此外,由于独占特性的存在,缺失块不会再加载到二级Cache中。

如果初始指令是一条load指令,由于指令/数据Cache和TLB都是对称的,数据地址会同时被送入数据Cache和数据TLB中,这和指令Cache访问很相似。不同的是,数据Cache有两个存储体,所以它能在不同的存储体寻址,支持并发的load和store。另外,数据Cache的缺失会产生写回牺牲块。在二级Cache的数据填满数据Cache的同时,牺牲数据从数据Cache中取出并送入牺牲缓冲中。

如果这条指令是store指令而不是load指令,当store指令发射时,会像load指令一样查询数据Cache。存储缺失会导致块填满到数据Cache中,这和load缺失非常类似,因为Cache分配策略是写操作分配。store指令在产生冲突之前不会更新。在此期间,存储位于load-store队列中,这个队列是处理器乱序控制机制的组成部分。它能支持44个项,并向执行单元转发结果。数据Cache由ECC纠错码保护,所以读-修改-写操作需要通过存储来更新数据Cache。这是通过在load/store队列中收集完整的块以及始终写整个块来实现的。

Opteron 存储层次结构的性能

Opteron性能如何?评估的一个重要指标是处理器因等待存储层次结构而耗费的那部分执行时间所占的百分比。主要部件有指令Cache和数据Cache、指令TLB和数据TLB以及二级Cache。可惜的是,对于Opteron这样的乱序执行处理器,分离出存储器等待的时间是非常困难的,这是由于一条指令的存储器停顿很可能被下一条指令的成功执行所掩盖。

图5.20所示为一系列程序的CPI和每千条指令的缺失率,这些程序包括类似商业数据库的TPC-C程序和SPEC2000程序。很明显,SPEC2000程序并没有从Opteron的层次化存储结构中获益,但mcf例外(由于其存储器使用量和访问模式的特点,SPEC给它取了个昵称叫“Cache buster”)。SPEC程序平均每条指令的指令Cache缺失率是0.01%~0.09%,数据Cache的缺失率是1.34%~1.43%,二级Cache的缺失率是0.23%~0.36%。商业测试对存储器层次结构有更多的要求,它们的单条指令缺失率分别是1.83%,1.39%和0.62%。

那么测试出Opteron的CPI如何呢?我们知道CPI的峰值是0.33,即每个时钟周期3条指令。对于SPEC2000程序,Opteron的每个时钟周期是0.8~0.9条指令,平均CPI为1.15~1.30。对于数据库

测试, 由于 Cache 和 TLB 的缺失率较高, 所以其 CPI 只有 2.57, 即每个时钟周期 0.4 条指令。TPC-C 类似测试程序中 CPI 降低两倍的情况表明, 对于服务器来说, 其微处理器的负荷远比桌面微处理器的负荷要大。图 5.12 评估了基值为 0.33 的 CPI 及其由于存储器或流水线暂停引起的性能下降。

测试程序	Avg CPI	每千条指令的缺失						
		lcache	Dcache	L2	ITLB L1	DTLB L1	ITLB L2	DTLB L2
类 TPC-C 程序	2.57	18.34	13.89	6.18	3.25	9.00	0.09	1.71
SPECint2000 total	1.30	0.90	14.27	3.57	0.25	12.47	0.00	1.06
164.gzip	0.86	0.01	16.03	0.10	0.01	11.06	0.00	0.09
175.vpr	1.78	0.02	23.36	5.73	0.01	50.52	0.00	3.22
176.gcc	1.02	1.94	19.04	0.90	0.79	4.53	0.00	0.19
181.mcf	13.06	0.02	148.90	103.82	0.01	50.49	0.00	26.98
186.crafty	0.72	3.15	4.05	0.06	0.16	18.07	0.00	0.01
197.parser	1.28	0.08	14.80	1.34	0.01	11.56	0.00	0.65
252.eon	0.82	0.06	0.45	0.00	0.01	0.05	0.00	0.00
253.perlbmk	0.70	1.36	2.41	0.43	0.93	3.51	0.00	0.31
254.gap	0.86	0.76	4.27	0.58	0.05	3.38	0.00	0.33
255.vortex	0.88	3.67	5.86	1.17	0.68	15.78	0.00	1.38
256.bzip2	1.00	0.01	10.57	2.94	0.00	8.17	0.00	0.63
300.twolf	1.85	0.08	26.18	4.49	0.02	14.79	0.00	0.01
SPECfp2000 total	1.15	0.08	13.43	2.26	0.01	3.70	0.00	0.79
168.wupwise	0.83	0.00	6.56	1.66	0.00	0.22	0.00	0.17
171.swim	1.88	0.01	30.87	2.02	0.00	0.59	0.00	0.41
172.mgrid	0.89	0.01	16.54	1.35	0.00	0.35	0.00	0.25
173.applu	0.97	0.01	8.48	3.41	0.00	2.42	0.00	0.13
177.mesa	0.78	0.03	1.58	0.13	0.01	8.78	0.00	0.17
178.galgel	1.07	0.01	18.63	2.38	0.00	7.62	0.00	0.67
179.art	3.03	0.00	56.96	8.27	0.00	1.20	0.00	0.41
183.equake	2.35	0.06	37.29	3.30	0.00	1.20	0.00	0.59
187.facerec	1.07	0.01	9.31	3.94	0.00	1.21	0.00	0.20
188.ammmp	1.19	0.02	16.58	2.37	0.00	8.61	0.00	3.25
189.lucas	1.73	0.00	17.35	4.36	0.00	4.80	0.00	3.27
191.fma3d	1.34	0.20	11.84	3.02	0.05	0.36	0.00	0.21
200.sixtrack	0.63	0.03	0.53	0.16	0.01	0.66	0.00	0.01
301.apsi	1.17	0.50	13.81	2.48	0.01	10.37	0.00	1.69

图 5.20 在 AMD Opteron 系统上分别运行类 TPC-C 数据库测试程序和 SPEC2000 测试程序所得的结果。因为 Opteron 采用乱序执行指令, 所以采用每千条指令成功提交情况下获得的数据

图 5.21 假设存储器层次结构中各层的缺失互不重叠, 也不和流水线执行相重叠, 所以流水线的停顿比例较低。使用这种计算方式, 计算的高于基值且有益于存储器的 CPI 占定点程序 (从 1% 的 eon 到 100% 的 vpr) 的 50%, 浮点程序 (从 12% 的 sixtrack 到 98% 的 applu) 的 60%。更进一步看, 大约 50% 的存储器 CPI (占总体的 25%) 来自于定点程序的二级 Cache 缺失, 大约 70% 的存储器 CPI (占总体的 40%) 来自于对浮点程序的二级 Cache 缺失。如前所述, 二级 Cache 缺失由于时间太长, 难以通过额外的操作将其隐藏。

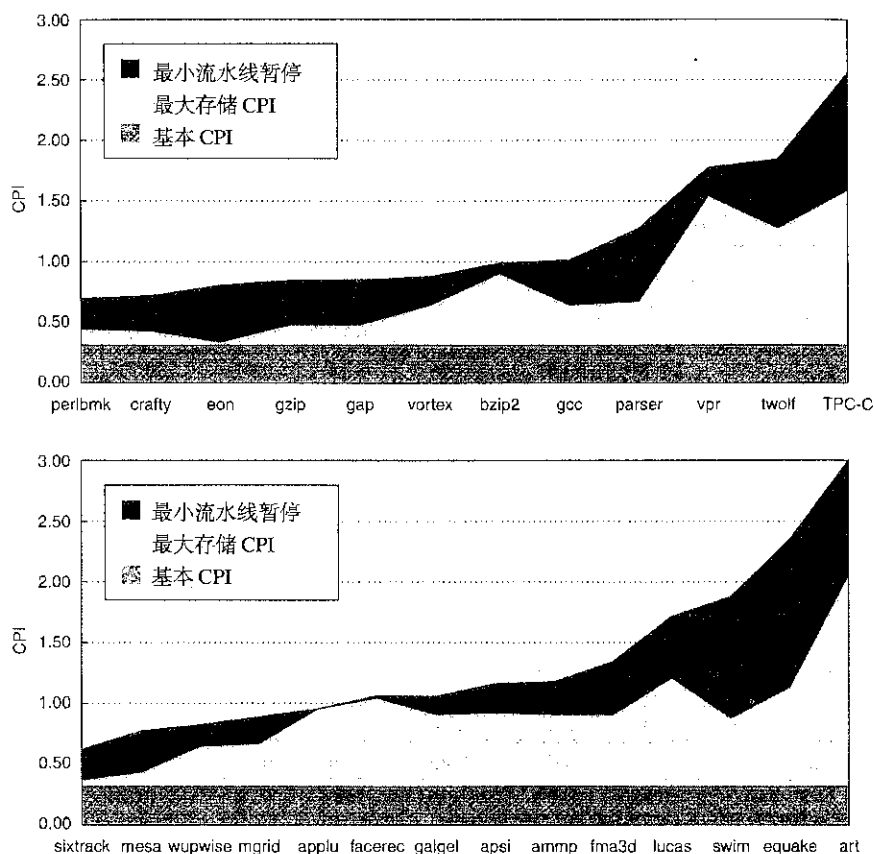


图 5.21 在上层使用 SPECint2000 程序（加上类 TPC-C 测试），在底层使用 SPECfp2000 程序，进行区域划分来评估基本 CPI、存储器暂停和流水线暂停的 CPI 衰减。它们按总 CPI 值由低到高排序。根据用每条指令在不同层次的缺失次数乘以对应层的缺失代价来估算存储器 CPI，从计算到的 CPI 中减去该存储器 CPI 和基本 CPI 得到流水线暂停 CPI。二级 Cache 缺失代价是 140 个时钟周期，所有其他的缺失都在二级 Cache 中命中。评估时假设没有存储和执行的相互重叠，故存储部分较高，因为其中某些的确和流水线暂停以及其他存储访问相互重叠了。由于 mcf 会因其 13.06 的 CPI 淹没其余数据，所以它不包含在此。存储缺失必然在 mcf 上重叠了，否则 CPI 会增至 18.53

最后，图 5.22 比较了 Opteron 和 Pentium 4 的数据 Cache 和二级 Cache 的缺失率，通过 10 个 SPEC2000 的测试指出每条指令的缺失率。尽管执行的是同样的指令系统编译出的程序，但编译器和结果代码序列会因存储层次结构的不同而不同。下表总结了两种不同的存储层次结构：

处理器	Pentium 4 (3.2 GHz)	Opteron (2.8 GHz)
数据 Cache	8 路组相联，16 KB，64 字节分块	2 路组相联，64 KB，64 字节分块
二级 Cache	8 路组相联，2 MB，128 字节分块，包含数据 Cache	16 路组相联，1 MB，64 字节分块，不含数据 Cache
预取	8 个数据流至二级 Cache	1 个数据流至二级 Cache

尽管 Pentium 4 有更高的相联度，但数据 Cache 比它大 4 倍的 Opteron 有更低的 Cache 缺失率。在几何标准偏差为 1.75 时，5 个 SPECint2000 程序在一级 Cache 缺失率比值的几何平均数为 2.25；对于 5 个 SPECfp2000 程序，在几何标准偏差为 1.72 时，几何平均数为 3.37（见第 1 章对几何平均数和标准偏差的回顾）。

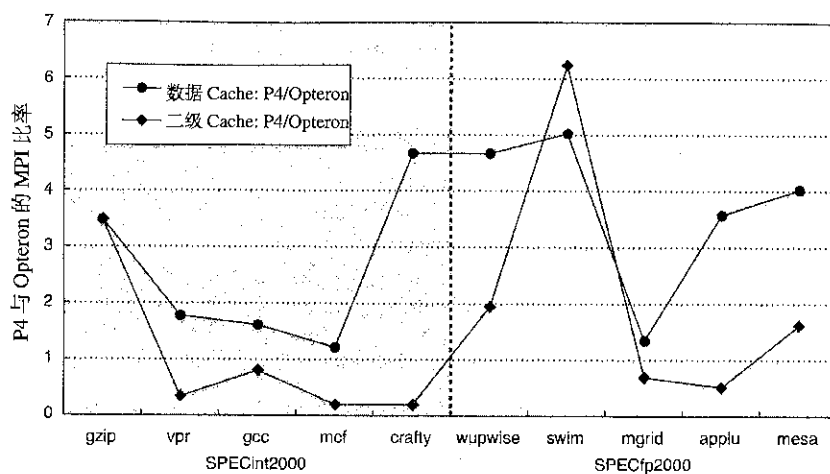


图 5.22 P4 和 Opteron 在每条指令缺失率上的比较。数值较大说明 P4 的缺失率较高。10 个测试程序中, 5 个是 SPECint2000, 5 个是 SPECfp2000 (两个处理器和其存储层次在上表中描述)。在标准偏差为 1.42 时, 5 个 SPECint 程序在两个处理器上性能比率的几何平均数为 1.00; 在标准偏差为 1.25 时, SPECfp 程序的性能几何平均数表明 Opteron 快 1.15 倍。注意, 实验中 Pentium 4 的时钟频率是 3.2 GHz, 其较高的时钟频率并不在此实验中表现出很好的可用性。图 5.10 表明有一半这类程序会因为 P4 的硬件预取而获益: mcf, wupwise, swim, mgrid 和 applu

Pentium 4 有比 Opteron 大 2 倍的二级 Cache 容量和块容量, 以及更好的预取技术, 所以它每条指令的二级 Cache 缺失比较少。但奇怪的是, Opteron 在 10 个测试程序中, 有 4 个都有更小的二级 Cache 缺失。这一差异表现在二级 Cache 缺失率比值的平均数和高标准偏差上: 对于定点程序来说, 几何平均数为 0.50, 标准偏差为 3.45; 对于浮点程序来说, 几何平均数为 1.48, 标准偏差为 2.66。正如早先提到的, 这个非直观的结果可能是由于不同的编译器和优化导致的。另一个可能的解释是, 由于 Opteron 较低的存储时延和较高的存储带宽有助于其硬件预取, 所以在许多浮点程序上, 它能减少缺失 (见图 5.10)。

5.7 谬误和易犯的错误

作为计算机系统结构中的定量原则, 存储层次结构看上去不会轻易受到谬误和易犯错误的影响。但实际情况却是大相径庭, 甚至于在这里限于篇幅, 我们都无法一一赘述。

谬误: 由一个程序的 Cache 性能推测另一个程序的 Cache 性能。

图 5.23 显示了当 Cache 容量变化时, 使用 SPEC2000 基准测试程序中的三个程序分别进行测试所获得的指令缺失率和数据缺失率。三个不同的程序, 对于 4096 KB 的 Cache, 每千条指令的数据缺失数分别是 9, 2 和 90, 对于 4 KB 的 Cache, 每千条指令的数据缺失数分别是 55, 19 和 0.0004。像数据库这样的商业程序即使使用很大的二级 Cache 仍然有很大的缺失率, 这与 SPEC 程序的情况完全不同。显然, 从一个程序的 Cache 性能推测另一个程序的 Cache 性能是不保险的。

易犯的错误: 模拟足够的指令以获得存储层次结构的精确性能指标。

这里实际有三个陷阱。第一, 用小规模的踪迹来预测一个较大 Cache 的性能; 第二, 在整个程序运行过程中, 局部性行为会有变化; 第三, 程序的局部性可能会随着输入而改变。

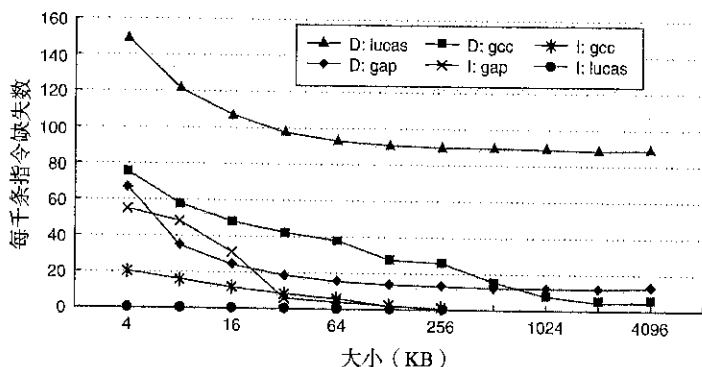


图 5.23 Cache大小从4 KB到4096 KB变化时,每千条指令和数据的缺失数。gcc指令缺失数是lucas的30 000~40 000倍,而lucas的数据缺失次数是gcc的2~60倍。gap, gcc和lucas来自于SPEC2000基准测试程序

图5.24显示了对于同一个SPEC2000程序,当输入不同时所累积的平均每千条指令缺失数。从这些输入中发现,其最开始的19亿条指令的平均存储器访问率和其后的指令的平均缺失率大不相同。

本书的第一版中包含了一个例子。由于踪迹存储器访问次数太少,导致强制缺失率却很高(如1%)。一个强制缺失率为1%的程序如果每秒访问存储器1000万次(本书第一版时的水平),那么它每秒将从存储器访问几百兆字节的数据:

$$\frac{10\,000\,000 \text{ accesses}}{\text{Second}} \times \frac{0.01 \text{ misses}}{\text{Access}} \times \frac{32 \text{ bytes}}{\text{Miss}} \times \frac{60 \text{ seconds}}{\text{Minute}} = \frac{192\,000\,000 \text{ bytes}}{\text{Minute}}$$

但是实际的缺页错误比率和进程大小表明上述推论并不成立。

易犯的错误: 在DRAM中过分强调存储器带宽

若干年前,RAMBUS在DRAM的接口上进行了创新。其产品Direct RDRAM可以用单片DRAM提供高达1.6 GB/s的带宽,这个峰值速度比通常的SDRAM快了8倍。图5.25比较了存储器模块和系统中各种版本的DRAM和RDRAM的价格。

计算机中大部分存储器访问都通过两级Cache结构完成,所以在不减少时延的情况下也无法知晓到底高带宽有多少作用。根据Pabst[2000]的研究,一台使用400 MHz DRDRAM的计算机和一台使用133 MHz SDRAM的计算机相比,它们运行办公软件时的平均性能是一致的。对于游戏,DRDRAM要快1%~2%。对于专业的图形应用,DRDRAM要快10%~15%。这些结果是在800 MHz的Pentium 3(集成256 KB二级Cache)上得到的,系统芯片组支持133 MHz总线和128 MB存储器。

一种衡量RDRAM成本的方法是晶片大小。对于相同容量的存储器,RDRAM比SDRAM的晶片要大20%左右。DRAM的设计者使用冗余的行和列,显著提高了DRAM芯片存储器的成品率,所以一个更大的接口可能对成品率有很大的影响,且影响并不是成比例的。成品率属于商业机密,但价格不是。按照图5.25中的数据推算,2000年时RDRAM的价格会比SDRAM高2~3倍,在2006年此比率也并未降低。

RDRAM对于那些存储器容量不大但带宽要求高的系统最为合适。比如Sony Playstation。

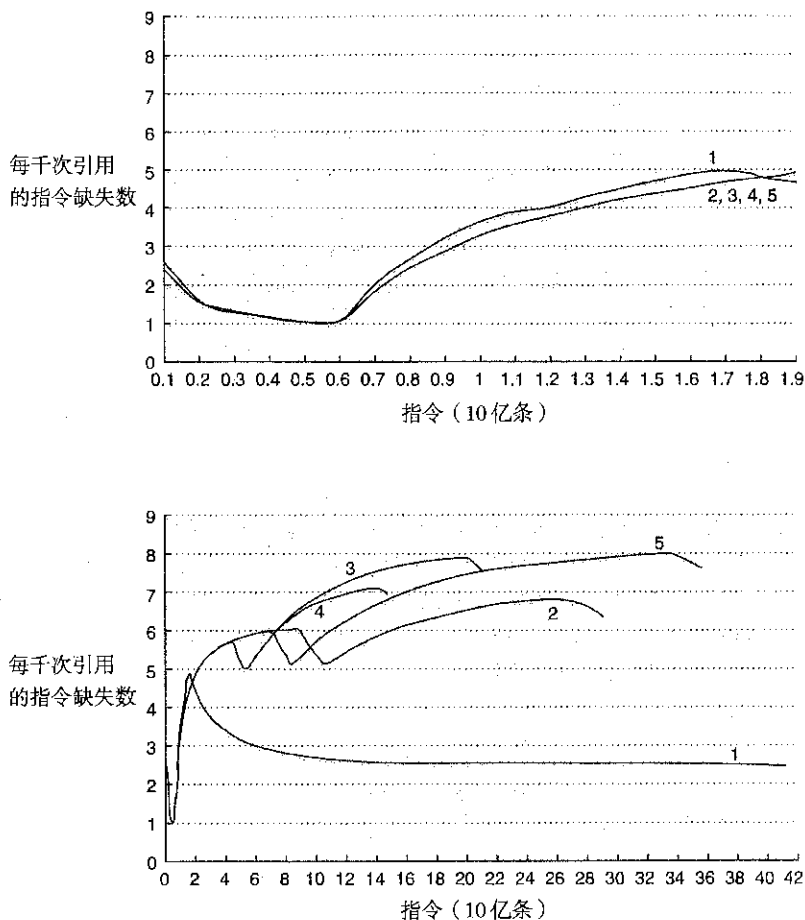


图 5.24 SPEC2000 perl 测试中, 5 个不同的输入得到 5 个不同的每千次引用的指令缺失数。前 19 亿条指令之前, 缺失率的变化并不大, 但是运行完后可以看出缺失率的变化以及它与输入的相关性。上图显示了前 19 亿条指令的平均缺失数, 5 种输入结果都近似, 开始是 2.5, 最后是 4.7 左右。下图显示了运行完成后的缺失率, 根据输入的不同, 其代码数量从 16 亿到 41 亿条不等, 它们的缺失率在 2.4~7.9 之间变化。图中的结果是在 Alpha 处理器上得到的, 它拥有分离的指令和数据一级 Cache, 都是 2 路组相联, 64 KB 大小, 采用 LRU 算法, 还有一个采用直接映射的 1 MB 的一体二级 Cache

易犯的错误: 基于 Cache 的系统不提供高带宽。

使用 Cache 可以减少存储器时延, 但是可能无法为依赖存储器的应用程序提供高带宽。对这样的应用程序, 系统结构设计者必须在 Cache 的后台设计高带宽的内存。NEC SX7 是个极端的例子, 它提供了多达 16 384 个 SDRAM 存储块。它是向量计算机, 存储器性能不依赖于数据 Cache (见附录 F)。图 5.26 显示了 2005 年使用 Stream 测试得到的最好的 10 个结果, 这些数据表明复制数据时可以达到的带宽 [McCalpin 2005]。毫无疑问, NEC SX7 处于顶级水平。

只有四种计算机的存储器性能依赖数据 Cache, 它们的存储器带宽比 NEC SX7 慢 7~25 倍。

ECC?	模块		Dell XPS PC					
			无 ECC			ECC		
	DIMM	RIMM	A	B	B-A	C	D	D-C
存储器还是系统?	DRAM		系统		DRAM	系统		DRAM
存储器容量 (MB)	256	256	128	512	384	128	512	384
SDRAM PC100	\$175	\$259	\$1519	\$2139	\$620	\$1559	\$2269	\$710
DDRDRAM PC700	\$725	\$826	\$1689	\$3009	\$1320	\$1789	\$3409	\$1620
价格比 DDRDRAM/SDRAM	4.1	3.2	1.1	1.4	2.1	1.1	1.5	2.3

图 5.25 2000 年, 存储器模块和系统中 SDRAM 和 DDRDRAM 的价格比较。没有 ECC 时, DDRDRAM 存储模块的价格是 DRAM 的 4 倍多, 有 ECC 时是 3 倍多。除了 128 MB 和 512 MB, 还有一种 384 MB 的 DDRDRAM 的价格是 DRAM 的 2 倍多。除了 DRAM 的带宽不同, 系统其他配置都相同。Dell XPS 计算机系列的配置除了存储器都是一样的: 800 MHz Pentium 3 处理器, 20 GB ATA 硬盘, 48 速 CD-ROM, 17 英寸显示器, 软件为 Microsoft Windows 95/98 和 Office 98。这些价格是 2000 年 6 月时 pricewatch.com 网站公布的最低价格。在 2005 年 9 月, 256 MB PC800 DDRDRAM 需要 76 美元, PC100 到 PC150 系列的 SDRAM 需要 15~23 美元, 前者贵了 3.3~5.0 倍 (2005 年 9 月时, Dell 不再提供仅仅是 DRAM 型号不同的系统, 所以我们还是采用 2000 年的数据)

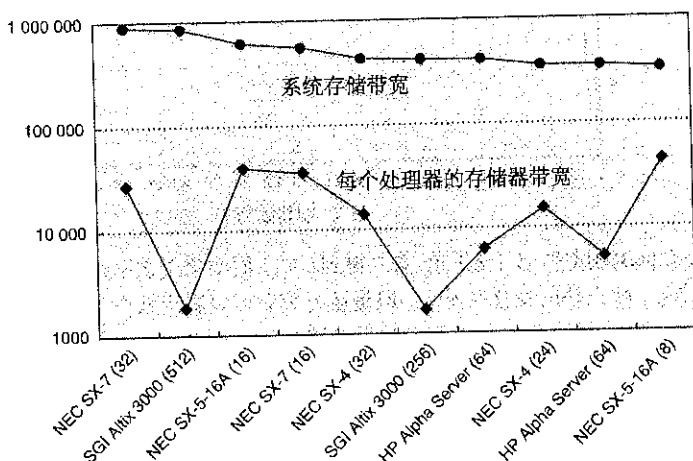


图 5.26 用流测试中的未裁减的数据副本来衡量存储器的带宽的十个最佳结果[McCalpin 2005]。处理器的数目在圆括号中标明。其中有两个基于 Cache 的集群 (SGI), 两个基于 Cache 的 SMP (HP), 但绝大多数是和 NEC 不同代和具有不同处理器数量的向量机。系统使用 8~512 个处理器来获得存储器高带宽。系统带宽指的是所有处理器的带宽和。把系统带宽除以处理器个数即可得到处理器带宽。流测试主要用在简单向量处理器上, 它其实是一个简单的综合测试程序, 用来衡量存储器的稳定带宽 (单位为 MB/s)。它比较适用于处理的数据量大于可用 Cache 容量的系统中

易犯的错误: 在不为虚拟化设计的指令集系统结构上实现虚拟机。

在 20 世纪 70 年代和 80 年代, 很多计算机系统设计者在设计时并没有特意去确保所有要读写有关硬件资源的指令都是特权指令。由于没有这些约束, 导致 VMM 不能在这些系统结构上使用, 其中包括 80x86, 这里我们就以它为例。

图 5.27 指出有 18 种指令由于这样的原因无法虚拟化[Robin 和 Irvine 2000]。其中有两大类指令是

- 在用户模式下读控制寄存器，从而暴露出虚拟机上运行的操作系统。
- 检查分段的系统结构所需的保护，但却假定操作系统在最高的特权级运行。

虚拟存储器也是要关注的问题，因为 80x86 和大部分 RISC 系统结构一样，TLB 不支持进程 ID 标注，VMM 和客户操作系统共享 TLB 会产生更大的开销，每个地址空间的改动都需要刷新 TLB。

问题种类	80x86 指令的问题
当运行在用户模式时，访问敏感寄存器无须 trap 中断	保存表寄存器全局描述符 (SGDT) 保存表寄存器本地描述符 (SLDT) 保存表寄存器中断描述符 (SIDT) 保存机器状态字 (MSW) 标志位入栈 (PUSHF, PUSHFD) 标志位出栈 (POPF, POPFD)
当在用户模式下访问虚拟存储器机制时，80x86 保护检查指令失效	从段描述符中载入访问权限 (LAR) 从段描述符中载入段的界限 (LSL) 如段描述符可读，进行读校验 (VERR) 如段描述符可写，进行写校验 (VERW) 段寄存器出栈 (POP CS, POP SS, ...) 段寄存器入栈 (PUSH CS, PUSH SS, ...) 调用不同的特权级 (CALL) 从不同的特权级返回 (RET) 跳向不同的特权级 (JMP) 软中断 (INT) 保存选段寄存器 (STR) 移入/出段寄存器 (MOVE)

图 5.27 虚拟化产生问题的 18 条 80x86 指令的概要[Robin 和 Irvine 2000]。上面的 5 条指令允许程序在用户模式下控制寄存器，而无须 trap 中断，例如表寄存器描述符。标志位出栈指令会修改包含敏感信息的控制寄存器，但在用户模式下将失效而无任何提示。80x86 体系中段的保护检查在下面的一组指令中，当读取控制寄存器时，作为该指令执行的一部分，都会隐式地检查特权级。进行检查时必须确保操作系统运行在最高特权级，但是对客户虚拟机并没有这样的要求。只有移入段寄存器操作会试图修改控制状态，但是，保护检查同样会阻止它这么做

虚拟化 I/O 也是 80x86 面临的重要挑战，部分原因是因为它同时支持存储器的 I/O 映射和独立的 I/O 指令，但更重要的原因是因为有数量巨大、种类繁多的设备和驱动程序，使得 VMM 无法管理。还有第三方设备厂商提供的专用驱动，可能不适合虚拟化。对于虚拟机实现来说，一个简单的方法是直接在 VMM 中加载一个实际的驱动。

为简化 VMM 在 80x86 上的实现，AMD 和 Intel 都提出了基于 80x86 的扩展系统结构。Intel 的 VT-x 为虚拟机运行提供了一个新的执行模式、一个面向虚拟机状态的系统结构定义、快速虚拟机切换的指令，以及一组用来选择调入 VMM 环境的参数。VT-x 在 80x86 中加入了 11 条指令。AMD 的 Pacifica 做了相似的改进。

在开启 VT-x 使能模式后(使用新的 VMXON 指令)，VT-x 给客户操作系统提供了四个特权级，这四级在优先级上低于原始的四级。在虚拟机控制状态 (VMCS) 下，VT-x 捕获了虚拟机所有的状态，并提供了原子指令来保存和恢复 VMCS。除了临界状态之外，VMCS 还包括一些配置信息来决

定何时调入 VMM 以及调入 VMM 的具体原因。为减少 VMM 被调入的次数,此模式加入了一些敏感寄存器的影子版本(shadow version),同时加入掩码以检查某些敏感寄存器的临界位是否会在 trap 中断前改变。为减少虚拟化虚拟存储器的开销,AMD 的 Pacifica 另外加入了一个间接层,称为嵌套页表,用它就不必再使用影子页表。

有趣的是,AMD 和 Intel 目前正在提出一个与此不同的新模式。如果 Linux 或 Windows 这样的操作系统在其内核中开始使用该模式,将会导致 VMM 的性能下降 100 倍。不过, Xen 计划使用 VT-x 来支持 Windows 作为客户操作系统。

5.8 结论

图 5.28 比较了面向桌面计算机和服务器的微处理器存储层次结构。它们的一级 Cache 都基本相似,主要的差别在于二级 Cache 的大小、晶片大小、处理器时钟频率和每个时钟的指令数。

MPU	AMD Opteron	Intel Pentium 4	IBMPower 5	Sun Niagara
指令集系统结构	80x86 (64 位)	80x86	Power PC	SPARC v9
应用范围	桌面	桌面	服务器	服务器
CMOS 工艺 (nm)	90	90	130	90
晶片大小 (mm ²)	199	217	389	379
每时钟发射指令数	3	3 RISC 操作	8	1
每个芯片的处理器数	2	1	2	8
时钟频率 (2006)	2.8 GHz	3.6 GHz	2.0 GHz	1.2GHz
每个处理器中的指令 Cache	64 KB	12000 RISC 操作	64 KB	16 KB
	2 路组相联	踪迹 Cache(96 KB)	2 路组相联	1 路组相联
L1 I 时延 (时钟周期)	2	4	1	1
每个处理器中的数据 Cache	64 KB	16 KB	32 KB	8 KB
	2 路组相联	8 路组相联	4 路组相联	1 路组相联
L1D 时延 (时钟周期)	2	2	2	1
TLB 条目数 (I/D/L2 I/L2 D)	40/40/512/512	128/54	1024/1024	64/64
最小页大小	4 KB	4 KB	4 KB	8 KB
片内二级 Cache	2 × 1 MB	2 MB	1.875 MB	3 MB
	16 路组相联	8 路组相联	10 路组相联	2 路组相联
L2 存储体数目	2	1	3	4
L2 时延 (时钟周期)	7	22	13	指令 22, 数据 23
片外三级 Cache	-	-	26 MB	-
	-	-	12 路组相联	-
L3 时延 (时钟周期)	-	-	87	-
块大小 (L1 I/L1 D/L2/L3, 字节)	64	64/64/128/-	128/128/128/256	32/16/64/-
存储器总线宽度(位)	128	64	64	128
存储器总线频率	200 MHz	200 MHz	400 MHz	400 MHz
存储器总线数目	1	1	4	4

图 5.28 2005 年桌面计算机和服务器的微处理器芯片及存储层次结构

各层的设计是相互影响的,设计者必须采取整体、系统的观点以做出明智选择。存储层次结构设计者们面临的挑战,是选择合理的参数使各层次协同工作,而不是发明新的技术。存储系统速度的提升远远跟不上高速处理器速度发展的步伐,从而出现了更多的选择:预取,基于 Cache 的编译器,还有更大的分页。在成本、性能、功耗和复杂性之间进行折中:避免在性能、功耗、硬件、设计时间和调试时间等方面的无用耗费。系统结构设计者通过精确、定量的分析方法来实现这个目标。

5.9 历史回顾和参考文献

在随书光盘的K.6节中,我们分析了Cache、虚拟存储器和虚拟机的发展史,在这些技术的发展过程中,IBM扮演了极其重要的角色,其中也包含了进一步阅读的参考文献。

5.10 范例分析及习题^①

范例分析 1: 通过简单的硬件实现 Cache 性能优化

通过这个示例阐明以下概念:

- 小而简单的 Cache。
- 路预测。
- 流水线 Cache。
- 多组 Cache。
- 合并写缓冲区。
- 关键字优先和提前重启动。
- 非阻塞 Cache。
- 在简单的有序处理器上计算 Cache 性能影响。

设想(并非实际情况)现在要构建一款简单的有序处理器,它对所有非数据存储访问指令只要一个CPI。在本例中,我们将分析小而简单的Cache、路预测Cache、流水线Cache、多组Cache、合并写缓冲区,以及关键字优先和提前重启动的性能表现。图5.29给出了通过执行所有基准测试程序得出的SPEC2000数据缺失率(每千条指令的缺失)。

D-cache misses/inst: 2,521,022,899,870 data refs (0.32899--/inst);					
1,801,061,937,244 D-cache 64-byte block accesses (0.23289--/inst)					
容量	直接映射	2路LRU	4路LRU	8路LRU	全相联LRU
1 KB	0.0863842--	0.0697167--	0.0634309--	0.0563450--	0.0533706--
2 KB	0.0571524--	0.0423833--	0.0360463--	0.0330364--	0.0305213--
4 KB	0.0370053--	0.0260286--	0.0222981--	0.0202763--	0.0190243--
8 KB	0.0247760--	0.0155691--	0.0129609--	0.0107753--	0.0083886--
16 KB	0.0159470--	0.0085658--	0.0063527--	0.0056438--	0.0050068--
32 KB	0.0110603--	0.0056101--	0.0039190--	0.0034628--	0.0030885--
64 KB	0.0066425--	0.0036625--	0.0009874--	0.0002666--	0.0000106--
128 KB	0.0035823--	0.0002341--	0.0000109--	0.0000058--	0.0000058--
256 KB	0.0026345--	0.0000092--	0.0000049--	0.0000051--	0.0000053--
512 KB	0.0014791--	0.0000065--	0.0000029--	0.0000029--	0.0000029--
1 MB	0.0000090--	0.0000058--	0.0000028--	0.0000028--	0.0000028--

图 5.29 SPEC2000 测试的数据缺失率(每千条指令的缺失)[Cantin 和 Hill 2003]

CACTI 是一个评估工具,用于访问时间、时钟周期、动态性、耗散功率、印制电路板上 Cache 芯片的面积大小和 Cache 组织的评估。当然对于给定的 Cache 组成,有很多种不同的电路设计,对于给定的印制电路板特征大小,也有很多不同的技术,但 CACTI 假设采用

^① 本范例分析由 Norman P. Jouppi 提供。

一般的组织形式和技术。这可能对某些特定的 Cache 设计和技术评估不是很精确，但在定量分析以不同大小组织的 Cache 性能时，这种方法还是相当精确的。CACTI 的网址为 <http://quid.hpl.hp.com:9081/cacti/>。假设所有的 Cache 缺失花费 20 个时钟周期。

- 5.1 [12/12/15/15]<5.2>下面的问题是使用 CACTI 研究小而简单的 Cache 性能，假设使用 90 nm 技术。
- [12] <5.2>比较按 64 字节分块的 32 KB Cache 和单一存储体的访问时间。相对于直接映射，2 路组相联 Cache 和 4 路组相联 Cache 的访问时间是多少？
 - [12] <5.2>比较按 64 字节分块的 2 路组相联 Cache 和单一存储体的访问时间。相对于 16 KB 的 Cache，32 KB 和 64 KB 的 Cache 的访问时间是多少？
 - [15] <5.2>对于典型的一级 Cache 组织形式，其访问时间是随着容量的大小增加的倍数是 B ？还是 B 的平方根？或是 B 的对数？
 - [15] <5.2>在图 5.29 中，对于给定的缺失率和 Cache 访问时间（0.90 ns），找出具有最小存储器访问时间的 Cache 组织形式。这样的 Cache 组织形式是什么？对于这个容量的 Cache，它是否有最低的缺失率？
- 5.2 [12/15/15/10]<5.2>假如现在正在研究路预测一级 Cache 将会带来的性能改进。假设一个 32 KB、2 路组相联、单一存储体的一级数据 Cache 限制了当前的时钟周期。作为一种 Cache 组织形式的选择，路预测 Cache 模拟了一个有 85% 精度的 16 KB 直接映射 Cache。如果在其他方面未做限定，假设一个不成功的预测访问命中 Cache 会多花一个时钟周期。
- [12] <5.2>与路预测 Cache 相比，当前 Cache 的平均存储器访问时间是多少？
 - [15] <5.2>如果所有其他的组件在更快的路预测 Cache 时钟周期下运行（包括存储器），使用路预测 Cache 对性能有什么样的影响？
 - [15] <5.2>路预测 Cache 通常只用于依赖指令队列或缓冲的指令 Cache 中。假设在数据 Cache 中使用路预测，有 85% 的预测精度，而且其后续操作（例如，其他指令的数据 Cache 访问、相关操作等）在假定预测正确的前提下被发射。这样一来，一个错误的路预测会迫使流水线停顿，并重新产生 trap 中断，这需要 15 个时钟周期。使用路预测的数据 Cache 之后，每条 load 指令平均访存时间的变化是正面的还是负面的？改变了多少？
 - [10] <5.2>作为路预测的一种替代方法，很多大容量的相联二级 Cache 将标注和数据访问串行化，仅需要激活所需的数据集阵列。这样节省了功耗，但增加了访问时间。使用 CACTI 详细的 Web 接口，针对 0.090 微米的处理器，1 MB 4 路组相联的 Cache，拥有 64 字节分块，144 位读出，1 个存储体，仅 1 个读/写端口和 30 位的标签。每次访问的动态读功耗比率，以及和并行访问相比，串行化标注和数据访问的访问时间比率分别是多少？
- 5.3 [10/12/15]<5.2>需要为一款新的微处理器评估使用流水线 Cache 和多组 Cache 相关性能的对比情况。假设 64 KB 的 2 路组相联 Cache 以 64 字节分块。流水线 Cache 由两条流水线组成，类似 Alpha 21265 的数据 Cache。多组 Cache 由两个 2 路组相联的 32 KB 存储体实现。使用 CACTI，并假设使用 90 nm 技术，回答下面的问题。
- [10] <5.2>和访问时间相比，Cache 的时钟周期是多少？Cache 使用了多少条流水线？
 - [12] <5.2>由于超细粒度的流水线操作引起缺失代价翻倍，以及流水线固有的数据相关性问题导致了 20% 的流水线停顿，在这种情况下的平均访存时间是多少？
 - [15] <5.2>如果每个时钟周期有一次访存，对存储体的访问呈随机分布（无须重排序），而存储体冲突会造成一个时钟周期的延迟。这种情况下，分组设计的平均访存时间是多少？

- 5.4 [12/15]<5.2>在一级 Cache 缺失时使用关键字优先和提前重启动,考虑将其用在二级 Cache 缺失上。假设 1 MB 的二级 Cache 以 64 字节分块,其重载通路为 16 字节宽。假设二级 Cache 每 4 个处理器时钟周期能写 16 字节,从存储器控制器中接收起始的 16 字节块需要 100 个时钟周期,其他从存储器中的每个块需花费 16 个时钟周期,数据能被旁路直接送入二级 Cache 的读端口。不考虑任何到二级 Cache 的缺失请求和一级 Cache 的数据请求传送时钟周期。
- [12] <5.2>分别在有或没有关键字优先和提前重启动时,计算二级 Cache 缺失需要花费多少时钟周期?
 - [15] <5.2>你认为关键字优先和提前重启动对于一级 Cache 和二级 Cache,哪个更重要? 和什么因素有关?
- 5.5 [10/15]<5.2>在写直达策略的一级 Cache 和写回策略的二级 Cache 之间,设计一个写缓冲区。二级 Cache 的写数据总线 16 字节宽,每 4 个处理器周期能对一个独立的 Cache 地址进行写操作。
- [10] <5.2>每个写缓冲区项需要多少字节宽度?
 - [15] <5.2>当使用 32 位存储指令来执行存储器清零时,如果所有其他指令都能在 store 指令执行时并行发射,并且块都在二级 Cache 中,那么在稳定状态下,使用合并写缓冲区与非合并的缓冲区相比,将获得什么样的性能加速?

范例分析 2: 通过先进技术优化 Cache 性能

通过这个示例阐明以下概念:

- 非阻塞 Cache。
- 编译器优化。
- 软硬件预取。
- 在更复杂的处理器上分析 Cache 性能影响。

矩阵内部行列交换的转置举例如下:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \Rightarrow \begin{bmatrix} A_{11} & A_{21} & A_{31} & A_{41} \\ A_{12} & A_{22} & A_{32} & A_{42} \\ A_{13} & A_{23} & A_{33} & A_{43} \\ A_{14} & A_{24} & A_{34} & A_{44} \end{bmatrix}$$

以下是实现该转置的一个简单的 C 程序:

```
for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
        output[j][i] = input[i][j];
    }
}
```

假设输入输出矩阵以行为主的顺序存储(行为主的顺序意味着行标签改变最快)。假设在一个拥有 16 KB 全相联(故不用考虑 Cache 冲突)、采用 LRU 替换、64 字节分块的一级数据 Cache 的处理器上,执行 256×256 的双精度转置。假设一级 Cache 的缺失或预取需要 16 个时钟周期,二级 Cache 总能命中,二级 Cache 每处理一个请求需要 2 个处理器时钟周期。假设数据在一级 Cache 中,内层循环的每次迭代需要 4 个周期,对于写缺失,Cache 采用写分配和写预取的策略,理想状况下,写回脏 Cache 块无须花费时钟周期。

- 5.6 [10/15/15]<5.2>对于上面给出的简单实现，输入矩阵的执行次序是不理想的。但是，使用循环交换优化方式又会在输出矩阵中产生不理想的次序。因为循环交换不能充分改善其性能，反而会被阻塞。
- [10] <5.2>需要用多大的块来填满单输入输出块的数据 Cache？
 - [15] <5.2>如果一级 Cache 是直接映射的，比较阻塞的和非阻塞 Cache 的缺失数关系。
 - [15] <5.2>使用 $B \times B$ 块的块大小 B 作为参数，写程序实现矩阵的转置。
- 5.7 [12]<5.2>假设为上面非阻塞的矩阵转置代码重新设计了硬件预取。最简单的硬件预取器只在缺失后预取连续的 Cache 块。更复杂的“非单元步幅”硬件预取器能分析出相关的流缺失，检测和预取超过一个步幅单元的块。与此对应，软件预取能很容易地确定非单元步幅和单元步幅。假设写预取直接写到 Cache 中，并且不被弄脏（在所需的数据被预取之前重写它）。在内层循环稳定的状态下，当使用理想的非单元步幅预取器时，其性能是怎样的（以每次迭代的时钟周期数衡量）？
- 5.8 [15/15]<5.2>假设为 5.7 的非阻塞矩阵转置代码重新设计硬件预取。不过在这里我们将对一种简单的、两条数据流的串行预取器进行评估。如果有两级访问通道可用，在一次缺失后，此预取器能取到 4 个连续的块，并把它们放到流缓冲中。流缓冲有空通道以负载平衡的方式访问二级 Cache。当一级 Cache 缺失，对最近一次缺失提供数据最少的流缓冲区被清空，并在新的缺失流中被重用。
- [15] <5.2>在内层循环稳定的状态下，当使用一种简单的、两条数据流的串行预取器时，如果假设性能由预取所限制，那么其性能如何（以每次迭代的时钟周期数衡量）？
 - [15] <5.2>在给定的二级 Cache 参数中，用于预取的百分比是多少？
- 5.9 [12/15]<5.2>使用软件预取时需要注意使预取随时可用，但是为了适应微系统结构的性能和将 Cache 被弄脏的程度减小到最低，也需要将此时预取的数量减小。而不同的处理器有不同的性能和局限性，这样就更增加了复杂性。
- [12] <5.2>修改上述非阻塞的代码，以实现软件预取。
 - [15] <5.2>非阻塞代码加入软件预取，所期待的性能改进是多少？

范例分析 3：存储器技术及优化

通过这个示例阐明以下概念：

- 存储系统设计：时延、带宽、价格和功耗。
- 分析存储系统性能表现。

根据图 5.14，考虑设计一种多样的存储系统。假设一款单芯片多处理器由 8 个 3 GHz 的内核组成，像 Opteron 一样，直接配备存储控制器（如完整的北桥芯片组）。这个多处理器芯片包含一块单独的共享式二级 Cache，有从这一级直接到存储器的缺失（即无三级 Cache）。图 5.30 是 DDR2 SDRAM 的时序图例。 t_{RCD} 是在一个存储体中触发一行所需的时间，CAS 延迟（CL）是在一行中读出一列所需的周期数。假设在一个标准的带有 ECC 的 DDR2 DIMM 上，RAM 有 72 条数据线。也假定每条数据线在 8 个脉冲长度下读出 8 位数据，即通过 DIMM 总共是 32 字节。假设 DRAM 每个页面的大小为 1 KB，有 8 个存储体， $t_{\text{RCD}} = \text{CL} \times \text{时钟频率}$ ，且时钟频率 = 每秒钟的传输量/2。发生 Cache 缺失时，通过一级和二级 Cache 命中而访问 DRAM 的片内时延是 20 ns。假设一款 1 GB DIMM 的 DDR2-667，CL = 5，其价格为 130 美元，而一款 1 GB DIMM 的 DDR2-533，CL = 4 的价格为 100 美元（见 <http://>

download.micron.com/pdf/technotes/ddr2/TN4702.pdf, 获取更多有关 DDR2 存储器组织形式和时序细节)。

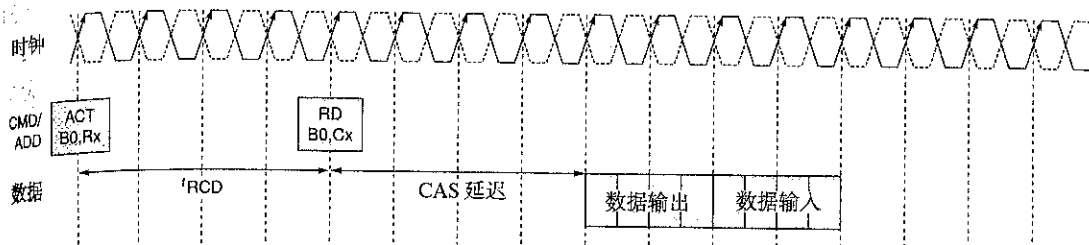


图 5.30 DDR2 SDRAM 的时序图

5.10 [10/10/10/12/12]<5.3>假设系统是桌面PC, 且CMP中只有单核可用, 假设只有一个存储器通道。

- [10] <5.3>如果使用 512 Mbit 的 DRAM, 在 DIMM 上有多少 DRAM? 如果每个 DRAM 仅仅只连接到每个 DIMM 数据引脚, 它必须有多少个数据 I/O?
- [10] <5.3>相对于 64 字节的二级 Cache 分块, 为支持 32 字节分块, 需要的脉冲长度是多少?
- [10] <5.3>在 DIMM 之间读一个活动的页面, 峰值带宽是多少?
- [12] <5.3>对于 1 GB DIMM 的 DDR2-533, CL = 4 的存储器, 从表示激活命令开始, 到 DRAM 中被请求的数据最后一位变成无效为止, 需要多少时间?

e. [12] <5.3>使用 DDR2-533 DIMM 的存储器, 相对于读取一个已经打开的页面, 读取一个需要激活的存储体的时延是多少 (请包括处理器处理缺失所需的时间)?

5.11 [15]<5.3>假设系统中只使用一个 DIMM, 其余的系统花费 800 美元。考虑使用 DDR2-667 和 DDR2-533 DIMM, 每 1000 条指令 3.33 次二级缺失的工作负载情况下的系统性能, 并假设所有的 DRAM 读操作都需要激活。假定所有 8 个内核作业量相同。如果一次仅有一个明显的二级缺失, 且 CPI 为 1.5 的有序执行内核中不包括二级 Cache 缺失的一次存储器访问时间, 那么当使用不同的 DIMM 时, 系统的整体性价比如何?

[12]<5.3>基于上面的系统提供一台服务器。CMP 上所有 8 核将会被 2.0 的 CPI 全部占用 (假设二级 Cache 缺失替换不会延迟)。理想情况下, 所有核缺失发生的时间均匀分布, 在每 1000 条指令 6.67 次二级缺失的工作负载下, 支持全部 8 核运行所需的带宽是多少?

[12]<5.3>大量 (超过三分之一) DRAM 的功耗都消耗在页面激活操作上 (见 <http://download.micron.com/pdf/technotes/ddr2/TN4704.pdf> 和 <http://www.micron.com/systemcalc>)。假设使用 1 GB 的存储器构建系统, 要么采用 4 存储体 512 Mbit \times 4 的 DDR2 DRAM, 要么采用 8 存储体 1 Gbit \times 8 的 DRAM, 两者速度相同。都采用 1 KB 的页面大小。假设 DRAM 处于加等待状态, 可以忽略功耗和等待到激活的时间。在低功耗方面, 哪一款 DRAM 要好些? 解释原因。

案例分析 4: 虚拟机

通过这个示例阐明以下概念:

虚拟机提供的性能。

虚拟化对性能的影响。

支持虚拟化的系统结构扩展的特点和影响。

Intel和AMD都对x86系统结构不能虚拟化的缺点进行了改进,扩展了x86系统结构。Intel的解决方案称为VT-x(虚拟化技术x86,见IEEE[2005]了解更多有关VT-x的内容),AMD的解决方案称为安全可视机(SVM)。Intel也有针对Itanium系统的相应技术:VT-i。图5.31列出了早期分别在本地执行、完全虚拟化和泛虚拟化上系统调用的性能,测试采用LMbench工具,在Itanium系统上使用Xen,用微秒级测量(经澳大利亚新南威尔士大学Matthew Chapman允许)。

测试程序	本地	完全虚拟化	泛虚拟化
Null call	0.04	0.96	0.50
Null I/O	0.27	6.32	2.91
Stat	1.10	10.69	4.14
Open/close	1.99	20.43	7.71
Install sighandler	0.33	7.34	2.89
Handle signal	1.69	19.26	2.36
Fork	56.00	513.00	164.00
Exec	316.00	2084.00	578.00
Fork + exec sh	1451.00	7790.00	2360.00

图 5.31 早期分别在本地执行、完全虚拟化和泛虚拟化上系统调用的性能

- 5.14 [10/10/10/10/10]<5.4>虚拟机为计算机系统提供了很多潜在的性能优势,例如它改善了总体拥有成本(TCO)和可用性。虚拟机能被用来提供以下性能吗?如果可以,如何实现?
- [10] <5.4>能简单地将大量运行在许多老式单处理器服务器上的应用程序,统一到一个单一的基于CMP的高性能服务器上吗?
 - [10] <5.4>能限制计算机病毒、蠕虫和木马对计算机的损害吗?
 - [10] <5.4>能为存储器需求大的应用程序提供更高的性能吗?
 - [10] <5.4>对于应用运行的高峰,能动态提供额外性能吗?
 - [10] <5.4>能在现在的机器上运行已有的旧操作系统平台的程序吗?
- 5.15 [10/10/12/12]<5.4>虚拟机会因为大量的偶然事件损失性能,例如执行特权指令、TLB缺失、trap中断和I/O。这些事件通常都由操作系统内核进行处理。因此,有一种评估程序运行在虚拟机上的性能损失的方式,即计算程序在系统和用户模式下执行所用的时间的比值。例如,在系统模式下花10%的时间执行的程序会由于运行在虚拟机上而减慢60%。
- [10] <5.4>当运行虚拟机时,执行哪种类型的程序会减慢得更多?
 - [10] <5.4>如果速度的减慢是系统时间的线性函数,根据上述减慢数据,当程序30%的执行时间花费在系统程序上时,它会减慢多少?
 - [12] <5.4>在完全虚拟化和泛虚拟化的情况下,图5.31中的函数减慢的均值是多少?
 - [12] <5.4>在图5.31中,哪个函数减慢得最少?原因是什么?
- 5.16 [12]<5.4>Popek和Goldberg对虚拟机的定义是:除了其性能外,虚拟机和真实机器很难区分。本题中我们将使用此定义,看能否找出程序是运行在本地机器上还是虚拟机上。Intel的VT-x技术有效地提供了第二套特权指令系统以便使用虚拟机。基于VT-x技术所给出的两组特权级,在虚拟机分别运行在本地计算机或另一个虚拟机的两种情况下,其相关性能会发生什么变化?

5.17 [15/20]<5.4>随着x86系统结构提供了虚拟化的支持,虚拟机的使用和发展成为了主流。比较 Intel 的 VT-x 和 AMD 的 SVM 虚拟化技术,AMD 的 SVM 详情可参考 http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf。

a. [15] <5.4>VT-x 和 SVM 如何处理特权指令?

b. [20] <5.4>对于存储器使用量大的应用程序,VT-x 和 SVM 如何提供高性能?

范例分析 5: 综合: 高度并行化存储系统

通过这个示例阐明以下概念:

- 理解存储系统设计的折中对机器性能的影响。

图 5.32 的程序可以用来评估存储系统的行为。关键是有精确的时序和规划使程序在不同存储层次间有序操作。图 5.32 是 C 程序。第一部分是过程,利用标准调用,通过 CPU 时钟得到用户精确时间。如果在其他系统中运行,此过程需要修改。第二部分是一个嵌套的循环,以不同的步幅和 Cache 大小读写存储器。此段代码运行了多次以获得精确的 Cache 时钟。第三部分测量了嵌套循环的时间开销,以便在全局测量时间中将其减去得到访问时间。结果输出到 .csn 文件中,以方便导入电子数据表格。根据所需要回答的问题以及需要测量的系统存储器的大小,读者可能对 CACHE_MAX 参数做出更改。在单用户模式下,或者至少在关闭其他程序的情况下运行此段程序,能得到更可靠的结果。图 5.32 的代码源自于 U. C. Berkeley 的 Andrea Cusseau 编写的一段程序,具体细节可在 Saavedra-Barrera[1992] 中找到。它针对现在的主流机器,做了大量的修改,能运行在 Microsoft Visual C++ 下。图 5.32 中的程序地址是物理地址,这在极少数使用虚拟地址缓存的机器上是存在的,例如 Alpha 21264。通常,在重新启动后虚拟地址立刻会指向物理地址,因此需要重启机器以便在结果中得到平滑的曲线。以下习题中,假设所有存储层次结构中的组件大小都是 2 的幂。假设在二级 Cache 中(如果存在),页面大小远远大于块大小,二级 Cache 的块大小大于或等于一级 Cache 的块大小。图 5.33 显示了程序的输出,列出了典型的容量大小。

5.18 [10/12/12/12/12]<5.6>参考图 5.33 中的程序结果示例:

- [10] <5.6>有几级 Cache?
- [12] <5.6>一级 Cache 的容量和块大小是多少?
- [12] <5.6>一级 Cache 的缺失代价是多少?
- [12] <5.6>一级 Cache 的相联度是多少?
- [12] <5.6>当数据大小等于二级 Cache 大小时,产生了什么效果?

5.19 [15/20/25]<5.6>修改图 5.32 中的代码以测量以下系统的特征。将实验结果绘图, y 轴为时间, x 轴为存储器步幅。使用对数作为坐标轴的单位,描绘出每种 Cache 容量的曲线。

- [15] <5.6>一级 Cache 采用的是写直达策略还是写回策略?
- [20] <5.6>存储系统是阻塞的还是非阻塞的?
- [25] <5.6>对于非阻塞的存储系统来说,能支持多少存储器引用?

5.20 [25/25]<5.6>在多处理器存储系统中,低级别的存储器层次结构不能被单个处理器占满,但是能被多处理器占满共同工作。修改图 5.32 中的代码,在同一时间运行多个副本,给出以下问题的答案:

- [25] <5.6>共享二级 Cache 或三级 Cache (若存在)需提供多大的带宽?
- [25] <5.6>存储器系统提供了多大的带宽?

```

#include "stdafx.h"
#include <stdio.h>
#include <time.h>
#define ARRAY_MIN (1024) /* 1/4 smallest cache */
#define ARRAY_MAX (4096*4096) /* 1/4 largest cache */
int x[ARRAY_MAX]; /* array going to stride through */

double get_seconds() { /* routine to read time in seconds */
    time64 t_ltime;
    _time64(&t_ltime);
    return (double) t_ltime;
}

int label(int i) { /* generate text labels */
    if (i<1e3) printf("%ldB", i);
    else if (i<1e6) printf("%ldK", i/1024);
    else if (i<1e9) printf("%ldM", i/1048576);
    else printf("%ldG", i/1073741824);
    return 0;
}

int tmain(int argc, TCHAR* argv[]) {
    int register nextstep, i, index, stride;
    int csize;
    double steps, tsteps;
    double loadtime, lastsec, sec0, sec1, sec; /* timing variables */

    /* Initialize output */
    printf("\n");
    for (stride=1; stride <= ARRAY_MAX/2; stride=stride*2)
        label(stride*sizeof(int));
    printf("\n");

    /* Main loop for each configuration */
    for (csize=ARRAY_MIN; csize <= ARRAY_MAX; csize=csize*2) {
        label(csize*sizeof(int)); /* print cache size this loop */
        for (stride=1; stride <= csize/2; stride=stride*2) {

            /* Lay out path of memory references in array */
            for (index=0; index < csize; index=index+stride)
                x[index] = index + stride; /* pointer to next */
            x[index-stride] = 0; /* loop back to beginning */

            /* Wait for timer to roll over */
            lastsec = get_seconds();
            do sec0 = get_seconds(); while (sec0 == lastsec);

            /* Walk through path in array for twenty seconds */
            /* This gives 5% accuracy with second resolution */
            steps = 0.0; /* number of steps taken */
            nextstep = 0; /* start at beginning of path */
            sec0 = get_seconds(); /* start timer */
            do { /* repeat until collect 20 seconds */
                for (i=stride; i!=0; i=i-1) { /* keep samples same */
                    nextstep = 0;
                    do nextstep = x[nextstep]; /* dependency */
                    while (nextstep != 0);
                }
                steps = steps + 1.0; /* count loop iterations */
                sec1 = get_seconds(); /* end timer */
            } while ((sec1 - sec0) < 20.0); /* collect 20 seconds */
            sec = sec1 - sec0;

            /* Repeat empty loop to loop subtract overhead */
            tsteps = 0.0; /* used to match no. while iterations */
            sec0 = get_seconds(); /* start timer */
            do { /* repeat until same no. iterations as above */
                for (i=stride; i!=0; i=i-1) { /* keep samples same */
                    index = 0;
                    do index = index + stride;
                    while (index < csize);
                }
                tsteps = tsteps + 1.0;
                sec1 = get_seconds(); /* - overhead */
            } while (tsteps<steps); /* until = no. iterations */
            sec = sec - (sec1 - sec0);
            loadtime = (sec*1e9)/(steps*csize);
            /* write out results in .csv format for Excel */
            printf("%4.1f, ", (loadtime<0.1) ? 0.1 : loadtime);
        }; /* end of inner for loop */
        printf("\n");
    }; /* end of outer for loop */
    return 0;
}

```

图 5.32 评估存储系统的 C 程序

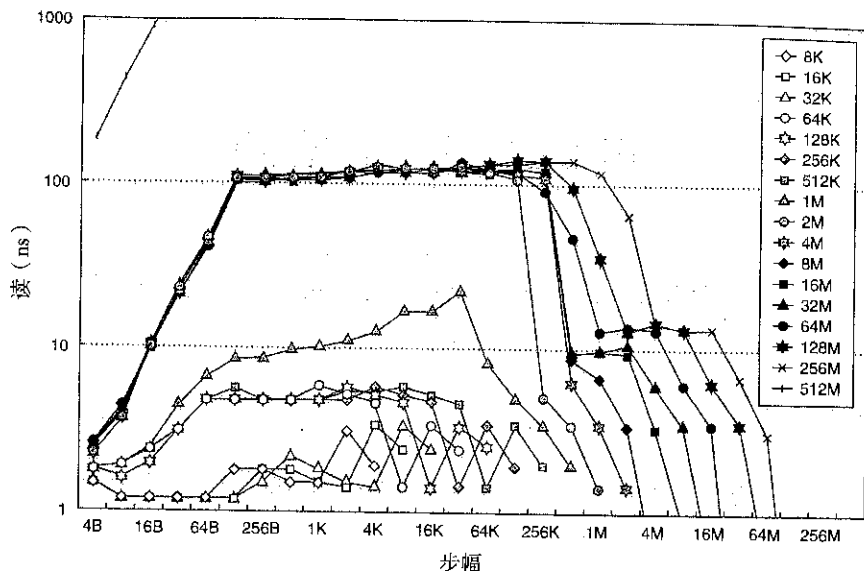


图 5.33 图 5.32 中的结果示例

5.21 [30]<5.6>既然指令级的并行能在有序的超标量处理器和预测 VLIW 上被有效地利用, 构建乱序 (OOO) 超标量处理器的一个重要原因是为了容忍由 Cache 缺失带来的不可预知的存储时延。因此, 可以考虑使支持 OOO 的硬件成为存储系统的一个组成部分。看 5.34 显示的 Alpha 21264 的图, 找出图中与 Cache 相对的定点和浮点队列及映射。队列调度指令的执行, 而映射将重命名寄存器指定。因此, 要支持 OOO, 还必须有其他必要的增加。21264 在芯片上只有一级数据和指令 Cache, 它们都是 64 KB、2 路组相联的。使用 OOO 超标量模拟器, 例如 SimpleScalar (www.cs.wisc.edu/~mscalar/simplescalar.html), 当队列和映射区域用在有序超标量处理器中附加的一级 Cache 区域时 (没有使用 21264 中的 OOO), 那么在存储密集的测试中性能将损失多少? 请确保机器其他部分尽可能相似, 这样才能保证对比的公正性。忽略由较大的 Cache 所造成的访问时间或时钟周期的增加, 以及大数据 Cache 对芯片布局的影响 (注意, 这里的对比不会完全准确, 因为代码不会由编译器调度给有序处理器)。

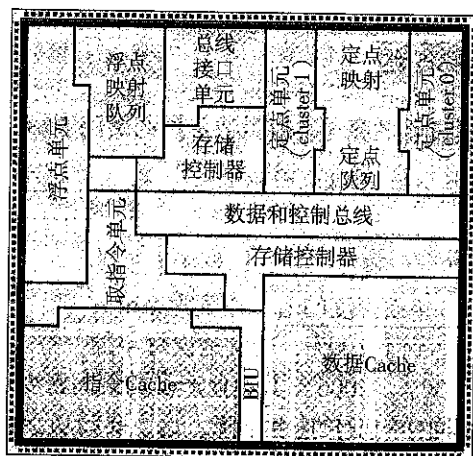


图 5.34 Alpha 21264 的平面图[Kessler 1999]

第6章 存储系统

我认为,硅谷的名字叫错了,如果你回顾近十年来产品的收入情况,磁盘比硅产品收入要多很多。因此应该把这个地方改名为氧化铁谷。

Al Hoagland
磁盘先驱者之一(1982)

将存储器及其传输带宽结合起来……实现了对日益增加的包含大量宝贵信息的磁盘和一个大宝库——互联网——的快速而可靠的访问……各种磁盘阵列容量的发展将远远超出其他计算机技术的发展。

George Gilder
“The End Is Drawing Nigh” Forbes ASAP[2000.4]

6.1 简介

像搜索引擎和电子拍卖之类的流行 Internet 服务增强了计算机 I/O 的重要性,因为任何人都会希望自己的 PC 能够访问互联网。I/O 重要性的增加也反映在我们时代名称的改变上。20 世纪 60 年代到 80 年代,称为计算机革命;20 世纪 90 年代称为信息时代,相对于原始的计算能力,映射出对信息技术的关注。Internet 服务依赖海量的存储和网络,前者是本章的内容,后者在附录 E 中涉及。

信息技术的关注点已从计算变为通信和存储,它强调可靠性、可测量性以及性价比。虽然人们可以容忍程序出现失败,但丢失数据是人们绝对不能接受的。因此,相对于其他计算机部件,存储系统需要更高的可靠性标准。可靠性是存储的基本要求,而且它也有自身成熟的理论基础——排队论。这是一个在响应时间和吞吐量之间进行折中的理论。决定使用处理器哪一部分功能的软件是编译器,而管理存储的则是操作系统。

6.2 磁盘存储的高级话题

磁盘厂商主要关注如何提高磁盘容量,容量通常用面密度表示,即每平方英寸的位数:

$$\text{面密度} = \frac{\text{磁盘表面磁道数}}{\text{英寸}} \times \frac{\text{磁道中的位数}}{\text{英寸}}$$

到 1988 年左右面密度以每年 29% 的速度递增,即每 3 年面密度增长一倍。从那以后直到 1996 年,增长速度达到了每年 60%,每 3 年面密度的增长速度为原来的 4 倍,相当于传统 DRAM 的增长率。从 1997 年到 2003 年,增长率达到了 100%,也就是说每年翻一番。后来由于没有新技术的出现,这个增长率最近又跌落回每年 30%。2006 年其商业产品的最高密度达到了每平方英寸 1300 亿位。每 GB 的价格以至少与面密度同样的增长速度下降,这主要归功于更小直径的驱动器的使用。从 1983 年到 2006 年间,每 GB 的成本下降了 100 000 倍。

磁盘在二级存储中的主导地位曾受到多次挑战。一个主要的原因是图6.1所示的磁盘和DRAM在访问时间上的差距。DRAM比磁盘快100 000倍,这种性能优势的代价是DRAM在每GB价格上比磁盘贵30~150倍。

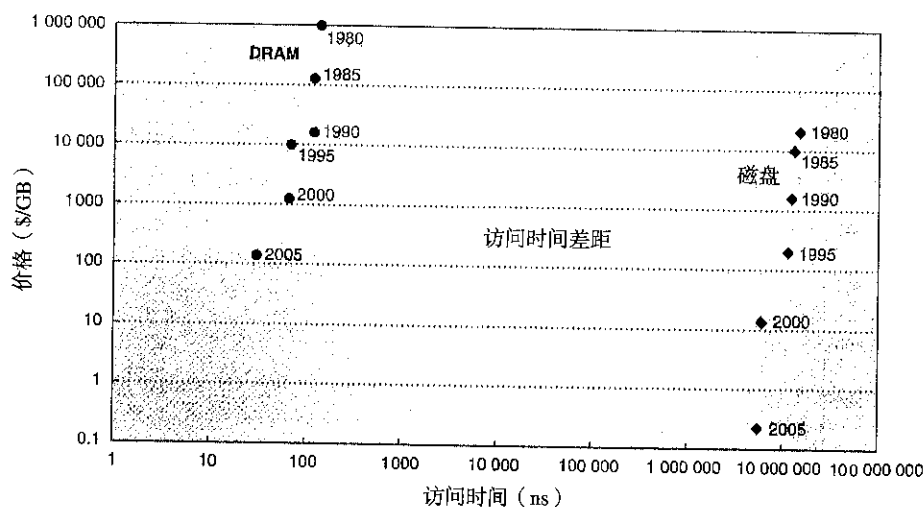


图6.1 1980、1985、1990、1995、2000和2005年DRAM和磁盘的价格与访问时间曲线。半导体存储器和磁盘在价格和访问时间上的差距引发了填补这一空缺的技术竞赛。随着磁盘和DRAM两者的发展,到目前为止这些尝试在产品化之前都失败了。在1990年到2005年之间,DRAM芯片的每GB价格几乎没有下降,而磁盘价格却大幅度下降

它们带宽上的差距更为复杂。以2006年一个较快速的磁盘为例,传输速率为115 MB/s,容量为37 GB容量,价格大约为150美元(见后面的图6.3)。而一个价格大约为300美元的2 GB的DRAM模块传输率是3200 MB/s(见第5章的图5.3),带宽比磁盘高28倍。但对DRAM来说,每GB的带宽高出了500倍,每1美元的带宽高出14倍。

许多人试图发明一种比DRAM便宜且比磁盘快的新技术来填补此空缺,但到目前为止,均以失败告终。目前还没有能够挑战磁盘二级存储地位的产品在市场上出现。根据先前的预测,即使有新产品问世,到那时DRAM和磁盘将会有长足的改进,价格也会相应降低,而这种新产品也不过是昙花一现。

近期出现的闪存是一种具有竞争能力的技术。这种半导体存储器是永久性的存储器,它和磁盘有同样的带宽,但时延却比磁盘快了100~1000倍。在2006年,闪存每GB的价格和DRAM一样。尽管闪存每GB的价格比磁盘贵50倍,但因为其体积和功耗都优于磁盘,所以它被广泛用于数码相机和便携式音乐播放器中。与磁盘和DRAM不同的是,闪存的每位只能重复擦写100万次左右,这也是其无法应用于桌面系统和服务器中的重要原因。

尽管在可预见的未来,磁盘仍会延续较高的可用性,但传统的扇区-磁道-柱面模型将会失效。该模型假设当访问同一个磁道上相邻块和同一个柱面中的块时,由于没有寻道时间,所以花费的时间较少;而且某些磁道可能要比其他的密集。

首先,磁盘开始提供高级智能接口,如ATA和SCSI,它们在磁盘中都包含一个微处理器。为加速连续单元的传输,这些高级接口对磁盘的组织更像是磁带而不是随机访问设备。逻辑块在单面上以蛇形排列,试图访问所有位密度相同的扇区(磁盘的记录密度是变化的,因为在技术上控制外层磁道上转速更快的块是很困难的,这可以用降低线性密度的方式使任务得到简化)。因此,连续

的块会分布在不同的磁道上。我们会在稍后的图 6.22 中看到在现代的磁盘中采用传统的扇区-磁道模型存在的问题。

其次,在磁盘中引入微处理器后不久,磁盘系统就加入了缓冲区用来缓存数据,直到计算机准备好接受为止,后来又引入了 Cache 以避免直接对它的读访问。它们都加入了命令队列,允许磁盘决定以哪种次序来执行命令,以在保持正确行为的同时获得最高性能。图 6.2 显示出队列深度为 50 时,通过优化访问调度,每秒钟 I/O 数能比随机访问 I/O 数翻一番。尽管在实际系统中,不太可能出现有 256 个命令在队列中的情况,但在这种情况下,每秒钟的 I/O 数能达到原来的三倍。如果考虑缓存、Cache、乱序访问等因素,实际磁盘的精确性能模型比扇区-磁道-柱面模型要复杂得多。

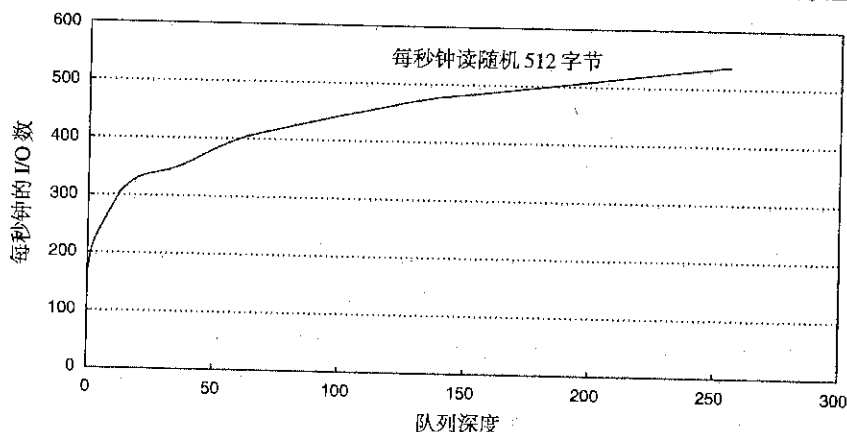


图 6.2 进行随机读 512 字节的操作，命令队列深度和吞吐量的对应关系。在开始没有命令队列时，磁盘每秒钟进行 170 次读操作，在有 50 个命令队列时，数量加倍，当达到 256 个命令队列时，变为三倍

最后,由于磁盘盘片的数量由过去的 12 片缩小到 4 片甚至 1 片,因此柱面的重要性大大降低了。

磁盘功耗

和处理器一样,磁盘的功耗也在不断增加。2006 年的典型 ATA 硬盘在其空闲时消耗 9 W,读写时消耗 11 W,寻道时消耗 13 W。由于磁道内圈的块更有效,所以磁盘直径越小就越能节省功耗。下面的一个公式给出了旋转速度、盘片大小与功耗的关系:

$$\text{功耗} = \text{直径}^{4.6} \times \text{rpm}^{2.8} \times \text{盘片数量}$$

因此,较小的盘片、较慢的转速和较少的盘片数都能降低磁盘电机的功耗,而电机功耗占了所有功耗的大部分比例。

图 6.3 指出 2006 年两款 3.5 英寸的磁盘规范。串行 ATA (SATA) 磁盘拥有较大容量和较低单位成本的优势,500 GB 的磁盘每 GB 的价格不到 1 美元。考虑到外形尺寸,SATA 使用最宽的盘片,总数有 4~5 片,但其转速为 7200 rpm,以通过较慢的寻道速度来降低功耗。对应的串行 SCSI (SAS) 驱动器主要针对性能的改善,转速达到 15 000 rpm,寻道更快。为降低功耗,盘片比起既定的外形尺寸要更窄,且只有一个盘片。这些因素导致了 SAS 的容量降低,只有 37 GB。

SAS 每 GB 的价格大约是 SATA 驱动器的五分之一多,反过来,SATA 每秒钟每个 I/O 的成本或每秒钟的 MB 传输率是 SAS 驱动器五分之一多。虽然 SAS 用了更少更小的盘片,但 SAS 的功耗却是 SATA 的两倍,这是由它更快的转速和寻道时间决定的。

	容量 (GB)	价格	盘片数	转速	直径 (英寸)	平均 寻道 时间 (ms)	功耗 (W)	I/O/s	磁盘 带宽 (MB/s)	缓存 带宽 (MB/s)	缓存 小大 (MB)	MTTF (hrs)
SATA	500	\$375	4或5	7200	3.7	8~9	12	117	31~65	300	16	0.6M
SAS	37	\$150	1	15 000	2.6	3~4	25	285	85~142	300	8	1.2M

图 6.3 2006 年 3.5 英寸的 SATA 与 SAS 参数比较。通过平均寻道时间加上一半的旋转时间再加上传输一个扇区 512 KB 的传输时延来计算每秒钟的 I/O 数

磁盘阵列的高级话题

磁盘阵列可以有效地提高存储系统的性能和可靠性。使用磁盘阵列的主要原因是可以使用多个磁盘驱动器和多个磁头臂，比一个大驱动器和一个磁头臂可以更大程度地提高磁盘潜在的吞吐率。数据呈带状分布的做法是将数据简单地分布到多个磁盘上去，这样如果数据文件很大，可以简单地实现同时访问几个磁盘（虽然阵列提高了吞吐率，但访问时延没有得到改进）。正如第 1 章提到的，磁盘阵列的缺点是多个设备导致可靠性降低： N 个设备的可靠性是一个设备的 $1/N$ 。

尽管当每个磁盘有相同的故障率时，磁盘阵列比数量较少的大容量磁盘更容易出错，但通过添加冗余磁盘可以提高可靠性。也就是说，如果单个磁盘发生故障，丢失的信息可以通过冗余信息重建。唯一的例外是在一个磁盘发生故障到修复这段时间（称为平均修复时间，MTTR）内，另一个磁盘也发生故障。因为磁盘的平均无故障时间（MTTF）是数十年，MTTR 通常用小时来计算，所以冗余可以使多个磁盘的可靠性比单个磁盘的高得多。

这种廉价磁盘冗余阵列简称 RAID，有人更愿意把它称为独立磁盘冗余阵列，字母 I 表示独立。由于具有恢复错误的能力和高吞吐量（要么以每秒钟的兆字节为量度，要么以每秒钟的 I/O 数为量度），使 RAID 更具有吸引力。加上体积小和小直径驱动器功耗低的优势，RAID 在大规模存储系统中占统治地位。

图 6.4 总结了 RAID 的 5 种标准级别，指出用户的 8 个磁盘如何在 RAID 的每一级上构成冗余或校验盘，并列出了每一级的优点与缺点。标准的 RAID 级别已经得到了很好的证明，这里我们只更深入地讨论高级别的 RAID。

- RAID 0：无冗余，昵称又叫 JBOD，即“只是一捆磁盘”，尽管数据是在磁盘阵列中带状分布的。这一级 RAID 通常扮演其他级别在成本、性能和可靠性上的衡量尺度。
- RAID 1：所有数据都有两个副本，又叫镜像或影像。这是最古老的磁盘冗余策略，但代价也是最高的。某些磁盘控制器为优化读性能，允许镜像磁盘做独立的读操作，但这种优化也意味着对镜像完全写完需更长时间。
- RAID 2：这种组织形式来自于存储器式错误纠正码在磁盘上的应用。之所以提及它，是因为最早的有关 RAID 的论文出现时，有这样一款磁盘阵列产品，但是后来不再使用了，因为其他形式的 RAID 更具有吸引力。
- RAID 3：因为更高级别的磁盘接口能了解每个磁盘的运行状况，故很容易发现哪块磁盘故障。设计者认为用另外一个磁盘存放数据的奇偶校验信息，当发生故障时，这个磁盘能起恢复功能。数据以带状形式存储，有 N 个数据磁盘，一个奇偶校验磁盘。当一个磁盘发生故障时，用该奇偶校验盘上的数据减去其他非故障磁盘的数据和，剩下的数据信息就是所丢失的信息（不管故障是发生在数据盘还是在奇偶校验盘）。RAID 3 假定在读或写时，数据在所有磁盘上都是带状分布的，当读写大量数据时，此方法非常有用。

的块会分布在不同的磁盘中，模型存在的问题。

其次，在磁盘

准备好接受为止

决定以哪种

时，通

出

(MB)

大小

(hrs)

MTTF

251

4 块交错奇偶校验	1个故障 1个校验磁盘
5 块交错分布式奇偶校验	1个故障 1个校验磁盘
6 行对角奇偶校验， EVEN-ODD	2个故障 2个校验磁盘

优点	缺点	生产该级别 RAID 的企业广泛使用
无空间开销	无保护	EMC, HP (Tandem), IBM
无奇偶校验计算; 快速恢复; 小数据的写速度高于其他 RAID 级; 快速读	最高的校验存储开销	未使用
不依赖故障磁盘, 自我诊断	以 2 为底的对数校验存储开销	存储概念
较低的校验开销; 对大块的读写有高带宽	不支持小数据量, 或随机读写	网络设备
较低的校验开销; 对小数据量的读有高带宽	奇偶校验磁盘是小数据量的写瓶颈	广泛使用
较低的校验开销; 对小数据量的读写有高带宽	小数据量的写→ 4 个磁盘访问	网络设备
针对 2 个磁盘故障的保护	小数据量的写→ 6 个磁盘访问; 2X 校验开销	

图 6.4 RAID 级别、容错能力以及冗余磁盘开销。目前很流行的一种数量化的分级方法是在引入术语 RAID 的一篇文章[Patterson, Gibson 和 Katz 1987]中提出来的。事实上, 非冗余磁盘阵列通常称为 RAID 0, 表示数据无冗余地呈带状分布在几个磁盘上。如果只是镜像中的一对磁盘中的一个出现故障, 那么本例中的镜像 (RAID 1) 最多可以承受 8 个磁盘故障; 最坏的情况是镜像中的某对磁盘同时出现故障。2006 年还没有 RAID 2 的商业化产品; 其他类型早已经出现在市场上。RAID 0+1, 1+0, 01, 10 和 6 是本书中要讨论的对象

- RAID 4: 很多应用程序呈现小数据量的访问特性。由于每个扇区有自身的差错检测, 可以通过允许每个磁盘执行独立的读操作来增加每秒钟的读操作次数。如果不得不读取所有磁盘来计算奇偶校验, 那么写操作还是很慢的。为增加每秒钟的写操作次数, 一种可选的办法是只使用两块磁盘。首先, 磁盘阵列读取那些将要被覆盖的数据, 在写入新数据之前计算哪一位会改变。然后读取奇偶校验盘上的旧数据值, 通过计算出的会改变的位来更新奇偶校验位, 然后将新的奇偶校验数据写入校验盘中。这些就是所谓的“小数据量写”, 它们仍然比“小数据量读”要慢——因为包含了对 4 次磁盘的访问——但它们仍然比每次写都访问所有的盘要快。RAID 4 的磁盘校验开销与 RAID 3 的一样低, 并且能在做大数据量的读写时和 RAID 3 一样快, 不过控制上更为复杂。
- RAID 5: 注意到在 RAID 4 中, 写小数据量的性能缺陷是必须读写同样的校验盘, 故这是一个性能瓶颈。为减小这种瓶颈, 将校验信息分布到所有阵列中的其他盘上。每条带上的奇偶校验块是轮转的, 以至于奇偶校验数据分布在所有磁盘上。磁盘阵列控制器在需要写入给定的块时, 必须能计算哪个磁盘上有奇偶校验数据, 不过这种计算较为简单。RAID 5 有与 RAID 4 和 RAID 3 一样低的磁盘校验开销, 它能像 RAID 3 一样做大数据量的读写, 像

RAID 4 一样做小数据量的读写, 而且与 RAID 4 相比, 有更高的小数据量带宽。然而和传统的 RAID 级相比, RAID 5 需要最新最复杂的控制器提供支持。

在简要回顾完所有经典的 RAID 级别之后, 我们现在着眼于自从 RAID 提出后, 被广泛使用的两个级别。

RAID 10 与 01 (或 RAID 1 + 0 与 0 + 1)

以下问题在 RAID 的文献中不常提及: 镜像与带状分布是如何相互作用的呢? 假设要存储 4 个盘的数据, 有 8 个盘可以使用, 是先把磁盘组成 4 对——以 RAID 1 的组织方式——再把数据带状分布于 4 对 RAID 1 中? 还是创建两个 4 磁盘系列——每个以 RAID 0 组织——之后再镜像写入? RAID 术语把前者称为 RAID 1 + 0 或 RAID 10 (带状镜像), 而把后者称为 RAID 0 + 1 或 RAID 01 (镜像带状)。

RAID 6: 多个磁盘故障的处理

基于奇偶校验的机制 RAID 1 至 5, 可使系统免于单个标识故障的发生。但是, 如果一个操作者在故障期间偶然错误地替换了磁盘, 则磁盘阵列将会发生两次故障, 数据将会丢失。另外一种担心是: 因为磁盘带宽的增长比容量增长更缓慢, 这导致 RAID 系统中单个磁盘的 MTTR 在增加, 而它又增加了二次故障的机会。例如, 在没有干扰的情况下, 500 GB 的 SATA 磁盘需要 3 小时才能连续读完。假设被损坏的 RAID 仍要继续提供数据服务, 而重组需要持续相当长的时间, 由此增加了 MTTR。除了增加重组时间之外, 另一种观点是: 在重组期间读取的数据越多, 发生无法校正的媒介故障的概率就越大, 从而导致数据丢失。关于并发性多故障的原因, 还有一种观点认为, 是磁盘阵列中磁盘数量的增加和大量使用比 SCSI 大而且慢的 ATA 硬盘的结果。

因此, 近年来如何解决多磁盘故障引起了人们的关注。例如, 网络设备就开始使用 RAID 4 构建文件服务器。对于用户来说双倍故障更为危险, 因此就产生了一个更强大的策略来保护数据, 称为行-对角奇偶校验, 即 RAID-DP [Corbett 2004]。类似标准的 RAID 策略, 行-对角奇偶校验在每个基本带上使用基于奇偶校验计算的冗余空间。为避免双倍故障, 在每条数据带上加入两个校验块。假设共有 $p + 1$ 个磁盘, 那么只有 $p - 1$ 个是数据盘。图 6.5 表示了 $p = 5$ 的情况。

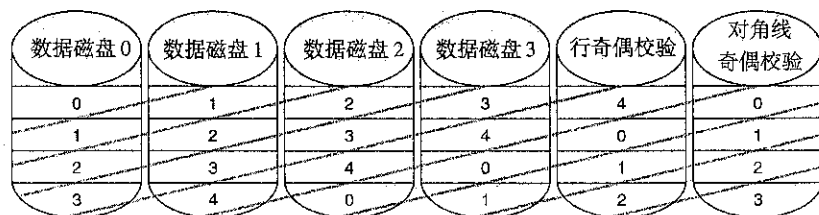


图 6.5 $p = 5$ 的行-对角线奇偶校验, 它能保护 4 个磁盘的双倍故障 [Corbett 2004]。该图显示了用来计算奇偶校验和存储在 1 个对角线奇偶校验磁盘中的对角线组。尽管这里给出的是 RAID 4 一样在单独的行奇偶校验磁盘和对角线奇偶校验磁盘上的校验数据, 但它有一个和 RAID 5 类似的行-对角奇偶校验轮换的版本。参数 p 必须是素数, 且要大于 2。但是在假设缺失磁盘所有数据位都为零的情况下, p 可以大于数据磁盘的数目, 而且策略仍然有效。这个变通使得向已有的系统添加磁盘变得容易了。NetApp 令 $p = 257$, 这突破了 256 个数据磁盘的上限。

行校验磁盘和 RAID 4 中的一样。它在数据带上包含了其他 4 个数据块的奇偶校验。每个对角奇偶校验磁盘的块包含了在同一对角线上其他奇偶校验块的偶校验信息。注意每个对角线都不覆

盖整个磁盘;例如,对角线0不完全覆盖磁盘1。因此需要正好 $p-1$ 个对角线来保护 p 个磁盘,故图6.5中的磁盘只有0到3共4条对角线。

我们假设图6.5中数据磁盘1到3故障,看看行-对角奇偶校验如何工作。我们无法使用首行和行奇偶校验来进行标准的RAID恢复,因为在磁盘1到3上缺失了两块数据。但是,我们能在对角线0上执行恢复,因为只是丢失了和磁盘3相关联的数据块。这样,行-对角奇偶校验首先使用对角线奇偶校验从故障磁盘上的4个块中恢复一个。因为每个对角线缺失一个磁盘上,所有的对角线缺失在不同的磁盘,因此存在两个对角线,它们缺失的是同一个块。在本例中它们是对角线0和对角线2,故接下来从故障磁盘1中的对角线2上开始恢复数据。一旦这些数据块被恢复,则标准的RAID恢复机制能被用来在标准RAID 4的数据带0和2上恢复2个更多的数据块,依次执行,恢复更多的对角线,直到2个故障磁盘完全恢复。

EVEN-ODD机制是早期IBM的研究人员开发的,类似于行对角奇偶校验,但其在操作和恢复过程中有多出一位的计算[Blaum 1995]。近期有文章指出了如何扩充EVEN-ODD以避免3个故障的破坏[Blaum 1996; Blaum 2001]。

6.3 实际故障的定义和实例

人们可以接受电脑死机,所有的程序重新启动,但是他们要求信息永远不会丢失。存储器最重要的任务就是不论发生什么事情,都保证存储的信息不会丢失。

第1章涵盖了基本的可靠性定义,本节对此进行了扩充,给出了其标准的定义,并介绍了故障的实例。

首先需要阐明引起混淆的术语。错误、差错和故障这些术语通常交替使用,但在可靠性文献中它们却有不同含义。例如,程序的出错到底是错误、差错,还是故障?它跟我们讨论的程序设计阶段或程序执行阶段有没有关系?如果该程序运行时没有出错,那它到底是错误、差错还是故障?来看另外一个例子。假设一个 α 粒子击中了—个DRAM存储器单元。如果这样没有改变存储器单元的值,那它是错误、差错还是故障?如果存储器无法访问已改变了的该位,那它是错误、差错还是故障?如果存储器具有纠错能力并把纠正后的值送给CPU处理,那它还会造成错误、差错或是故障吗?现在读者应该已经对这个问题的复杂程度有所了解。显然,为了更好地讨论这些问题,我们需要更准确的定义。

为了避免不准确的定义,以下使用的术语引自[Laprie 1985]与[Gray和Siewiorek 1991],它们已被IFIP工作组和IEEE计算机技术容错协议认定。我们的讨论把一个系统视为一个模块,当然这些术语也同样适用于那些子模块。我们从可靠性的定义开始:

计算机系统的可靠性(dependability)用于表示提供服务的质量,这里“可靠性”可用“信任”(reliance)代替。这种由系统提供的服务就是本地用户和与当前系统用户交互的其他系统感知到的实际行为。每个模型也是理想化的规范行为,在模型中,服务的规范性是对被期望行为的一致描述。系统故障(failure)的原因在于实际行为与规范行为发生偏差。故障的原因在于差错(error),即模型的缺陷。差错的原因在于错误(fault)。

错误发生时引发一个隐藏的差错。当服务启动时这个差错将开始起作用,直到它确实影响到所提供的服务时,故障就会发生。差错发生和故障出现之间的时间段称为差错时延。因而差错是系统错误的表现,而故障则是服务差错的表现。

让我们回到上面的例子中。程序中的错误称为**错误**，其结果是产生软件中的**差错**（隐藏的差错）。当服务启动时，差错开始作用，当由差错造成的错误数据影响到所提供的服务时，就发生了故障。

α 粒子击中了一个DRAM可以视为一个错误。如果它改变存储器中的数据，就产生一个差错。错误一直隐藏到该存储器字被读出。如果这个字影响了所提供的服务，故障就发生了。如果使用ECC纠错，故障就不会发生。

操作人员的失误也是一种错误，其结果是造成数据改变，这当然是一个差错。它一直隐藏直到服务启动，随后的过程和上面讨论的相同。

下面将阐述错误、差错和故障三者间的关系：

- 一个错误产生一个或多个隐藏的差错。
- 差错的特点是：(1) 当服务启动时才起作用；(2) 在隐藏和有效两个状态间循环；(3) 现行差错经常会在模块间传播而造成新的差错。因此一个现行差错是先前隐藏的差错或是其他的错误传播造成的。
- 当差错影响到所提供的服务时，就产生了组件故障。
- 这些特点是递归的，并适用于系统中任何一个模块。

Gray 和 Siewiorek 按照错误发生的原因，将其分成4类：

1. 硬件错误：设备发生故障，例如可能的 α 粒子击中存储器单元。
2. 设计错误：多为软件错误，少量是硬件设计错误。
3. 操作错误：操作和维护人员的失误。
4. 环境错误：火灾、洪水、地震、停电和蓄意破坏。

故障往往根据其持续时间分为短暂故障、间歇故障和永久故障[Nelson 1990]。短暂故障仅在很小一段时间存在，且不会复发。间歇故障导致系统在有故障和无故障之间震荡。永久故障不会随时推移而自我修复。

现在我们已定义了错误、差错和故障的区别，下面将观察实际系统中的例子。现实中很少公布系统中实际的差错发生率，这有以下两个原因：一是研究人员很少有机会对真正的大型硬件设备进行接触。二是业界几乎不允许研究人员发布故障信息，以免影响其所在公司的市场销售。下面介绍的是几个例外。

伯克利 Tertiary Disk 系统

在加州大学进行的伯克利 Tertiary Disk 系统项目为旧金山的 Fine Arts 博物馆构建了一个艺术品档案资料图片的服务器。这个数据库由7万多件艺术作品的高质量图片组成。它们被保存在一个通过以太网连接的20台PC机和368块磁盘组成的集群上。该集群的机柜有7英尺高。

图6.6给出了Tertiary Disk各部分的故障率。在构建这个系统之前，设计者假定数据磁盘是系统中最不可靠的部分。因为它们是机械的，又数量众多。紧随其后的是IDE磁盘，因为它们的数量和数据磁盘相当。接下来依次是电源和集成电路。系统中像电缆这样的无源设备被认为是几乎永远不发生故障的。

组件	总数	总故障次数	故障百分比
SCSI	44	1	2.3%
SCSI	39	1	2.6%
SCSI	368	7	1.9%
IDE/ATA	24	6	25.0%
磁盘驱动器盒底板	46	13	28.3%
磁盘驱动器盒电源	92	3	3.3%
以太网控制器	20	1	5.0%
以太网交换机	2	1	50.0%
以太网线缆	42	1	2.3%
CPU/ 主板	20	0	0%

图 6.6 18 个月运转过程中 Teritary Disk 系统各部件的故障率。对于每一种部件，表中均有它在系统中的总数、故障数和故障率。因为磁盘驱动器盒可能发生底板集成电路故障和电源故障，所以它在表中有两项。由于每个驱动器盒有两个电源，因此一个电源发生故障不会产生不利影响。这个集群由 20 台 PC 组成，装在一个 7 英尺高 19 英寸宽的机柜里，包含 368 个 8.4 GB 7200 转的 3.5 英寸 IBM 磁盘。每个 PC 机由 P6-200 MHz 的处理器和 96 MB 的 DRAM 组成，使用 FreeBSD 3.0 操作系统，使用 100 Mb/s 的以太网连接。所有的 SCSI 磁盘都由双端 SCSI 链连接两台 PC，支持 RAID 1。集群运行的主要应用程序为 Zoom Project，1998 年它是世界上最大的艺术品图片数据库，包括 72 000 件艺术品的图片。参见[Talagala 2000]

图 6.6 中打破了前面提到的一些假设。由于设计者听从制造商的建议使磁盘驱动器盒减少了振动，并且有良好的散热，结果是数据磁盘是非常可靠的。与此相反，PC 中容纳的 IDE 磁盘的底座却没有提供同样的环境控制（IDE/ATA 磁盘不用于存储数据，而是辅助应用程序和操作系统启动 PC）。图 6.6 显示 SCSI 的底座、电缆和以太网线缆还没有数据磁盘可靠。

因为 Teritary Disk 系统有大量的冗余部件，所以它有能力应付大范围的故障。部件被连接在一起，图片的镜像被存储起来，因此任何单个的故障都不会造成图片丢失。这种策略尽管开始时看来有些过分，但事实证明是很重要的。

这个实验还揭示了短暂故障和硬故障的区别。在概念上，图 6.6 中所有故障第一次出现时都是短暂故障，但是是更换部件还是继续使用的决定则是由操作员做出的。实际上故障一词没有被使用；设计人员借用了通常用在公司处理员工时使用的术语，即由操作员决定是否“开除”故障部件。

Tandem

这是一个业界的例子。Gray[1990]搜集了 Tandem 公司的计算机的故障数据。这家公司是容错计算方面的先驱，其主要应用是在数据库方面。图 6.7 中的曲线表示了 1985~1989 年间引起系统故障的差错的发生情况，即每个系统的绝对差错数量和差错发生率。数据表明硬件和维护的可靠性有显著的提高。1985 年 Tandem 计算机磁盘需要 Tandem 公司年检，后来用不需要定期检查的磁盘替代了原来的磁盘。1989 年芯片和连接器的减少以及软件容错能力的提高，使硬件造成的故障率下降到 7%。当硬件出错时，嵌入其中的软件常常是主要原因。数据表明 1989 年软件是系统失效的主要原因（62%），系统操作仅占 15%。

不过这些统计数字存在一个问题：它们都是依据报告数据得到的。例如，环境引起的故障（如停电等）未在报告中提到，因为这些被认为是与系统无关的因素。操作失误方面的数据也很难收集到，因为这要依赖操作员报告自己人为错误，而这会这影响到经理对他的看法和及加薪问题。

Gary认为以上两种差错的报告资料不完整。他的研究表明,如果想得到更高的可用性,要提高软件质量和软件容错能力、简化操作以及加强对操作的容错能力。

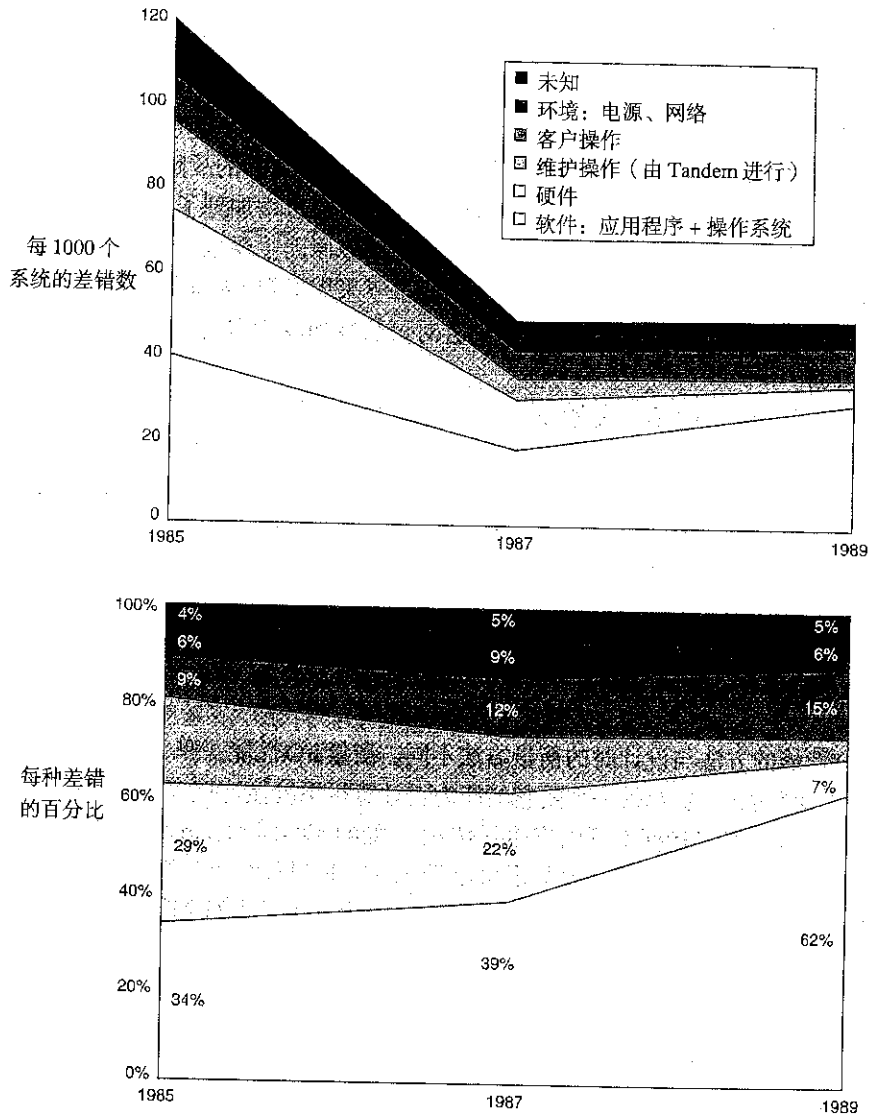


图 6.7 1985~1989 年 Tandem 计算机的差错。Gray[1990]根据用户的故障报告搜集了容错Tandem计算机的有关数据

其他方面研究: 操作员在可靠性中扮演的角色

Tertiary Disk 和 Tandem 是面向存储可靠性研究的两个实例, 我们需要在存储之外找出能更好地衡量人在故障中角色的方法。为了提高有关操作员错误数据的准确性, Murphy和Gent[1995]使系统在启动时自动提示本次重新启动的原因。他们把连续出现的系统崩溃归结为操作员错误, 还概括了哪些操作员行为会直接导致系统崩溃, 例如给参数错误的赋值、错误配置和错误的应用程序安装。尽管他们仍然认为操作员的差错报告率低于实际发生率, 但是他们的数据的精确性却大大高于Gray的数据, 它是采用由操作员填写表格后逐级上交的方法得到的结果。VAX 系统中的硬件和操作系

统引起的故障百分比从1985年的70%下降到1993年的28%，与此同时，由操作员引起的故障百分比从15%上升到52%。Murphy和Gent预测系统管理将是未来可靠性最重要的挑战。

最后这组数据来自美国政府。联邦通信委员会（FCC）要求所有的电话公司当他们出现至少30 000个用户或30分钟的故障中断时必须提交一份故障说明。这些详细的中断报告没有像早期的报告那样，受到操作员自行报告机制所带来的负面影响，因为故障原因是由调查人员确定而不是由设备操作人员自己确定的。Kuhn[1997]研究了1992—1994年间的中断原因，Enriquez[2001]继续研究了2001年上半年的情况。近年来，尽管在由于网络超载而引起的故障方面有了重大的改善，但由于人为因素造成的用户中断故障占总故障时间的比例从1/3增加到了2/3。

这四个实例和其他数据表明，今天大型系统中的故障原因主要是人为因素。硬件错误由于系统中的芯片和连接器数量的减少而下降，硬件可靠性由于采用了像RAID和ECC一类的容错技术而得到提高。至少一部分操作系统在加入新特性前会对可靠性加以考虑。因此，2006年故障的主要原因发生在其他方面。

尽管故障可能由于操作者的差错而产生，但系统在维护和升级过程中更容易出错。当今大部分存储器厂商宣称用户在磁盘生命周期内管理它的花费要超过购买磁盘的费用。因此，未来的可靠性的主要问题来自两个方面：容忍操作员错误和简化系统管理任务以避免错误。注意，RAID 6在用户错误地替换了好的磁盘的情况下，仍能维持存储系统的正常工作。

到此，我们已经介绍了可靠性技术的基础，给出了定义、示例研究和改进方法。后续内容将讨论存储的性能。

6.4 I/O性能可靠性评测

衡量I/O性能的方法与设计时的衡量方法不同。衡量I/O性能，一个是它的多样性：计算机上能连接什么类型的I/O设备？另一个是它的能力：计算机系统能连接多少I/O设备？

另外，传统性能衡量指标也可以应用到I/O上，如响应时间和吞吐率（I/O吞吐率也称为I/O带宽，响应时间也称为时延）。从下面两幅图可以看出响应时间和吞吐率相互影响的情况。图6.8所示为简单的生产者-服务器模型。生产者创建要执行的任务，并将其放到缓冲区；服务器从先进先出（队列）缓冲区中取出任务并开始执行。

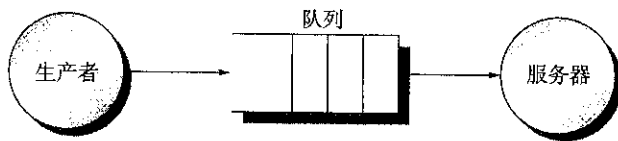


图 6.8 响应时间和吞吐率的传统生产者-服务器模型。响应时间是指一个任务从被放入缓冲区到被服务器执行完之间的时间间隔。吞吐率指的是单位时间内服务器执行完的任务数

响应时间定义为从一个任务被放入缓冲区到被处理完成之间的时间间隔。吞吐率是指单位时间内平均能完成的任务数。为达到尽可能高的吞吐率，服务器不应闲置，因此缓冲区不应为空。但从另一方面来看，响应时间也包括任务在缓冲器中的等待时间，因此如果缓冲区内的任务数越少，那么响应时间就会越短。

另一个衡量I/O性能的标准是I/O操作对处理器运行的干扰。传输数据会影响其他进程的运行。处理器处理I/O中断也要花费一定的代价。我们关心的是处理器将额外花费多少时钟周期去处理其他进程的I/O操作。

吞吐率与响应时间

图 6.9 是一个典型的 I/O 系统响应时间与吞吐率（或时延）的关系图。图中曲线表明，吞吐率少量提高会引起响应时间快速增长；相反，响应时间少量缩短会引起吞吐率快速降低。

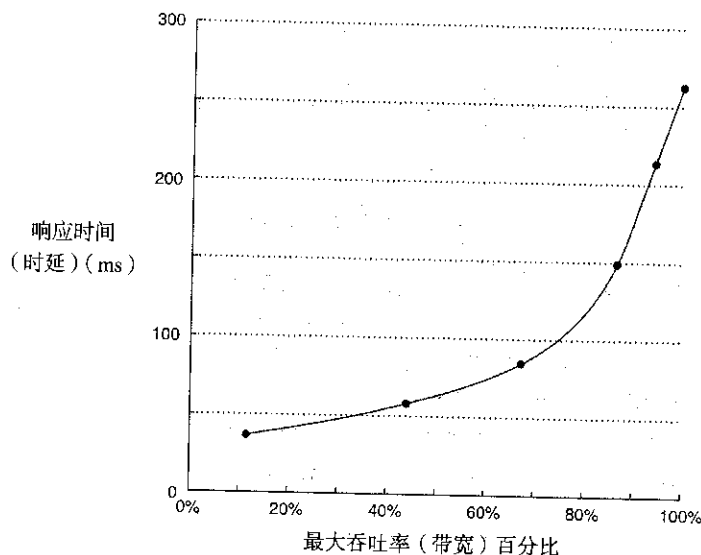


图 6.9 吞吐率与响应时间。时延通常表示为响应时间。注意最小的响应时间只能使吞吐率提高 11%，若要使吞吐率达到 100%，则对应的响应时间是最小响应时间的 7 倍。单从这条曲线的变化是不能说明问题的：为了描绘出准确的曲线，需要变化负载（并发）。Chen 等 [1990] 收集了有关这些磁盘阵列的数据

设计者如何在这些相互冲突的需求中寻求折中方案呢？图 6.10 给出的答案是人机交互。同计算机的一次交互分为三部分：

1. 进入时间：即用户输入命令所需的时间。
2. 系统响应时间：即从用户输入命令到显示响应结果所需时间。
3. 用户反应时间：即从接收响应结果到用户开始输入下一条命令的时间。

这三部分的时间之和称为交互时间。研究表明，用户工作效率与交互时间成反比。图 6.10 表明减少响应时间确实减小了交互时间：将响应时间减少 0.7 s，则传统交互时间会节省 4.9 s (34%)，图形交互时间会节省 2.0 s (70%)。这一令人难以置信的结果是符合人的特性的：快速的响应减少了人的反应时间。尽管这些研究已有 20 年之久，但直到今天，尽管处理器的速度已经提高 1000 倍，但响应时间通常长于 1 s，例如启动桌面计算机的应用程序，这是由于大量磁盘 I/O 或点击 Web 链接的网络延迟造成了长时间延迟。

为反映响应时间对用户生产率的重要性，I/O 基准测试程序提供了对于响应时间与吞吐率相互影响的新视角。图 6.11 给出了 3 种基准程序的响应时间要求，最大吞吐量都是在这样的限定条件下计算得到的，或者是 90% 的响应时间必须小于某一上限，或者是平均响应时间必须小于某一上限。

接下来介绍这些基准程序的细节。

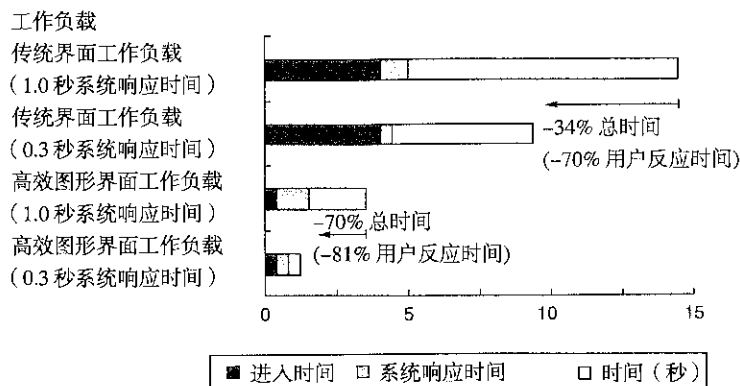


图 6.10 用户与计算机的一次交互对于传统系统和图形系统来说分成进入时间、系统响应时间和用户反应时间三部分。进入时间与系统响应时间无关。传统系统进入时间是 4 s，图形系统进入时间是 0.25 s。响应时间的减小造成总交互时间减小的额度比响应时间减小的额度还要大（数据取自 Brady [1986]）

I/O 基准程序	响应时间限定	吞吐量度量标准
TCP-C: 复杂队列 OLTP	≥ 90% 的事务应满足响应时间限制；大多数事务为 5 s	每秒钟交互的请求
TPC-W: 交互式	≥ 90% 的 Web 交互应满足响应时间限制；大多数 Web 交互为 3 s	每秒钟 Web 交互
Web 基准程序	平均响应时间 ≤ 40 ms	每秒 NFS 操作
SPECsfs97		

图 6.11 三种 I/O 基准程序的响应时间限定

事务处理基准测试程序

事务处理（TP 或 OLTP 即联机事务处理）主要关心 I/O 频率（每秒磁盘访问次数），它与数据传输率（每秒传输数据的字节数）不同。TP 通常包括对大量共享信息的改动，且 TP 系统保证发生故障时可以做出正确的动作。例如当一个客户在 ATM 机上取钱时，银行计算机出现故障，TP 系统应保证在客户拿到款项时记入账目，以及当客户未拿到钱时不改变账目。航班预订系统与银行系统一样都是 TP 的传统客户。

正如第 1 章提到的，TP 团体的 20 多个成员共同完成了一个工业基准测试程序，并发布了匿名报告[Anon 等 1985]。这个报告加速了事务处理协会的成立，这个组织从成立至今发布了 8 个基准测试程序。图 6.12 对这 8 个程序进行了总结。

基准测试程序	数据大小 (GB)	性能指标	首个结果发布日期
A: 借方信贷 (不再使用)	0.1~10	每秒事务数	1990 年 7 月
B: 批量借方信贷 (不再使用)	0.1~10	每秒事务数	1991 年 7 月
C: 复杂查询 OLTP	100~3000 (最小为 0.07 × TPM)	每秒新订单事务数	1992 年 9 月
D: 决策支持 (不再使用)	100, 300, 1000	每小时查询数	1995 年 12 月
H: 特殊决策支持	100, 300, 1000	每小时查询数	1999 年 10 月
R: 业务报告决策支持 (不再使用)	1000	每小时查询数	1999 年 8 月
W: 事务性 Web 基准测试	≈ 50, 500	每秒 Web 交互数	2000 年 7 月
App: 应用服务器和 Web 服务基准测试	≈ 2500	每秒 Web 服务交互数 (SIPS)	2005 年 6 月

图 6.12 事务处理基准测试程序。总结的结果包括性能尺度和在该性能尺度的性价比。TPC-A, TPC-B, TPC-D 和 TPC-R 已经不再使用

下面通过介绍 TPC-C 来了解这些基准测试程序的特点。TPC-C 使用数据库来模拟一个批发的订单入口环境,包括登记和发送订单、支付记录、检查订单状态和监控仓库存货水平。它同时运行 5 个具有可变复杂度的并发事务;数据库中包含 9 张表,其中记载大小可变的记录和用户。TPC-C 可用每分钟的事务数 (tpmC) 和系统价格进行度量。系统价格包括硬件、软件和三年的维护支持费用的总和。第 1 章的图 1.16 描述了顶层系统的性能和 TPC-C 的性能代价。

这些 TPC 基准测试程序具有以下特点,这些特点是 TPC 基准测试程序首先使用的,其中一些方面现在仍是独有的:

- 在基准测试的结果中包括价格因素。硬件、软件和三年系统维护协议的费用包括在所提交的报告中,这使得评价结果有助于提高性能和性价比。
- 为了适应吞吐量的增加,数据集大小可变。基准测试程序模拟一个实际的系统。系统中需求和存储数据的大小是同步增加的。例如,每分钟让上千人访问上百个银行账号是毫无意义的。
- 测试结果是经过审核的。在结果提交前,要经过 TPC 审核者的认证。审核者执行 TPC 的规则,保证只提交合理的结果。测试结果可能要受到质疑并且只有当争端解决之后才决定由 TPC 发布。
- 吞吐率是性能的尺度,但响应时间是受限的。例如,在 TPC-C 中,90% 的新订单事务的响应时间不能超过 5 秒。
- 第三方对基准测试程序进行维护。TPC 收来的款项付给一个权威组织的管理机构。该组织负责解决争端、处理有关基准测试程序变化等事宜。

SPEC 系统级文件服务器、邮件及 Web 的基准测试程序

SPEC 基准测试程序除了可以描述处理器性能特性之外,也可以应用于测试文件服务器、邮件服务器和 Web 服务器。

名为 SFS 的综合基准测试程序是由 7 家公司共同通过的,它可以用来评价运行 Sun Microsystems 的网络文件服务 NFS 的系统的性能。这一基准测试程序已升级至 SFS 3.0 (也称为 SPEC SPS97_R1) 以支持 NFS 第三版,该基准使用 TCP 及 UDP 作为传输协议,并使系统的操作更接近实际应用。基于 NFS 系统的测试程序包含了读、写和文件操作。SFS 提供了用于性能比较的默认参数。如半数的写操作在 8 KB 大小的块中执行,另一半写操作分别在大小为 1 KB, 2 KB 或 4 KB 的块中执行,而 85% 的读操作在全部块中进行,15% 的读操作在部分块中进行。

与 TPC-C 一样, SFS 依据吞吐率来对存储数据的数量进行调整:每秒执行 100 个 NFS 操作,规定容量就须增加 1 GB;同时对平均响应时间也有限制,这种情况下为 40 ms。图 6.13 给出了两个 NetApp 系统的平均响应时间和吞吐率的关系。很遗憾,与 TPC 基准测试程序不同, SFS 不能把不同的价格配置标准化。

SPECMail 是用来评价互联网服务提供商的邮件服务器性能的基准测试程序。SPECMail2001 基于标准 Internet 协议 SMTP 和 POP3,它测量用户数从 10 000 增长到 1 000 000 时的吞吐率和用户响应时间。

SPECWeb 是用来评价万维网服务器性能、测试并发访问用户数的基准测试程序。SPECWeb2005 的工作负载是通过模拟对 Web 服务提供者的访问体现的,该服务提供者的服务器包括几个组织的主页。它有三种工作负载:银行业务 Banking (HTTPS)、电子商务 E-commerce (HTTP 和 HTTPS) 和支持业务 Support (HTTP)。

时间和用
图形系统
的额度还

度标准
交互的请求
Web 交互

S 操作

),它与数据
系统保证发生
障,TP 系统
与银行系统

发布了匿名
了 8 个基准测

结果发布日期

年 7 月
年 7 月
年 9 月

年 12 月
年 10 月
年 8 月
年 7 月
年 6 月

TPC-A,

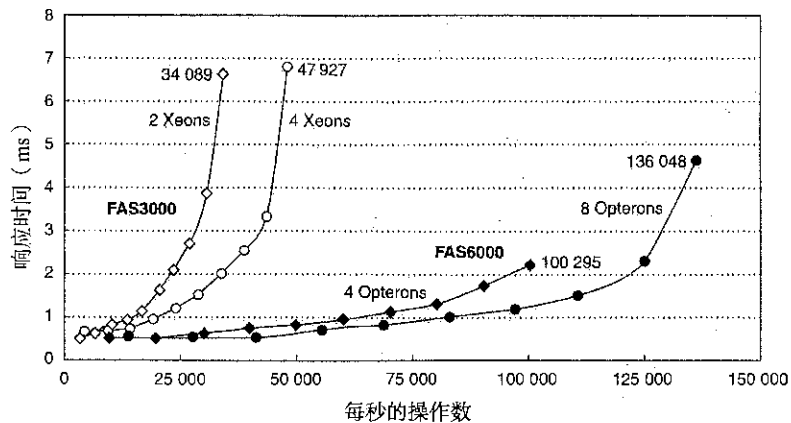


图 6.13 SPEC SFS97_R1 在两种配置的 NetApp FAS3050c NFS 服务器上的性能。在 2 个和 4 个处理器条件下分别为 34 089 和 47 927 ops/s。根据 2005 年 5 月的报告, 这些系统使用的都是 Data ONTAP 7.0.1R1 操作系统, 2.8 GHz 的 Pentium Xeon 微处理器, 每个处理器 2 GB 的 DRAM 存储器, 每个系统有 1 GB 的固定存储器, 168 块 15K RPM 72 GB 的光纤通道硬盘。这些硬盘由 2 个或 4 个 QLogic ISP-2322 FC 磁盘控制器连接

基准测试程序可靠性的实例

TPC-C 基准测试程序实际上具有可靠性的要求。基准测试程序系统应具备处理单一磁盘故障的能力。这意味着所有的提交者在存储系统中都使用某种 RAID 组织结构。

近期的成果集中在系统容错的有效性上。Brown 和 Patterson[2000]提出, 可以通过观察把错误注入系统后系统服务质量标准随时间的变化情况来度量系统的有效性。对于一个 Web 服务器来说, 重要的衡量标准是性能(用每秒请求的满足数量衡量)和容错程度(用存储子系统和网络拓扑结构等部分的容错数量衡量)。

最初的实验是在系统中注入一个错误——如磁盘扇区写错误——并记录系统反映在服务质量指标上的行为变化。实例比较了 Linux, Solaris 和 Windows 2000 Sever 提供的 RAID 软件实现工具。SPECWeb99 被用来提供工作负载并测量性能。RAID 软件卷中的一块 SCSI 磁盘被换成了仿真磁盘以便注入错误。仿真磁盘是一台有专用的 SCSI 控制器的运行软件的 PC。它把 PC、控制器和软件合并在一起。在 SCSI 总线上的其他设备看起来, 它就是一块磁盘。仿真磁盘允许注入错误。注入的错误中包括各种短暂的磁盘错误, 例如可纠正的读错误, 和永久性错误, 如磁盘媒介写入失败。

图 6.14 显示了各个系统在不同错误下的表现。上面的两幅图分别表示 Linux (左) 和 Solaris (右) 的情形。如果在完成重建前第二块磁盘出现故障, RAID 系统将丢失数据, 重建时间 (MTTR) 越长, 可用性越低。提高重建速度意味着降低应用性能, 因为重建会占用应用程序的 I/O 资源。因此, 需要在以下两种策略中做出选择: 提高重建速度, 尽管重建时系统性能会降低; 或是延长重建时间, 以保证重建时的系统性能, 但会缩短预定的平均无故障时间 MTTF。

虽然被测试系统都没有提供重建策略的有关文档, 但是通过错误注入就可以窥探到其内部策略。实验表明, 当磁盘发生故障不能使用时, Linux 和 Solaris 会在另一个正在工作的设备上自动开始重建。虽然 Windows 支持 RAID 的重建, 但是需要人工参与。因此, 如果无人干预, Windows 系统无法在系统第一次故障时重建数据, 这使得系统对第二次故障变得非常不可信, 这也增加了 Windows 的不稳定性。不过在有人工干预的情况下, 修复也能很快完成。

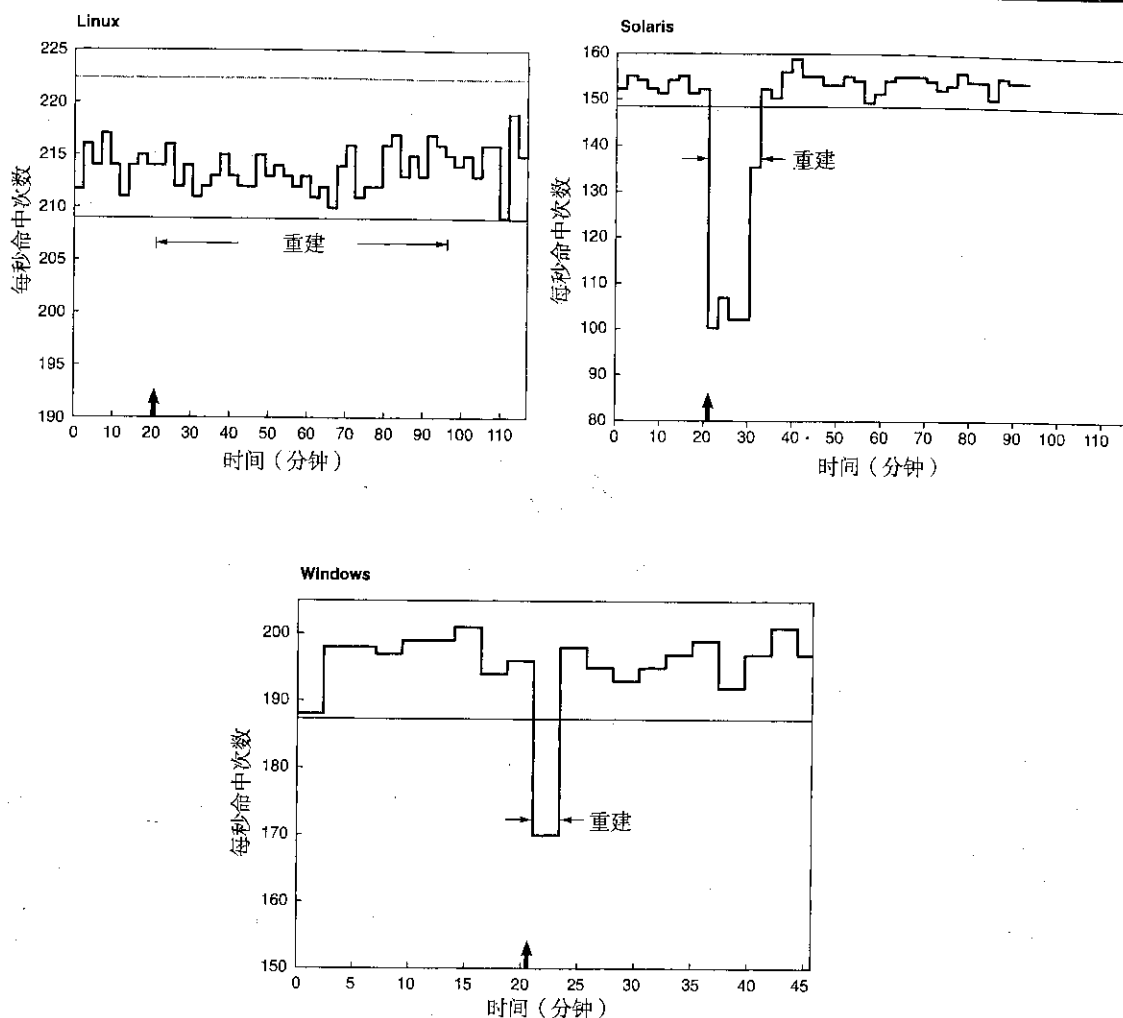


图 6.14 在同样的计算机上分别运行 Red Hat 6.0 Linux, Solaris 7 和 Windows 2000 操作系统, 有效性基准测试程序测出的对应软件 RAID 系统的结果。注意策略上的不同导致了 Linux 相对于 Windows 和 Solaris 的重建速度不同。y 轴表示运行 SPECWeb99 每秒命中的情况, 箭头表示植入错误的时间。顶端横线表示在错误插入前性能的 99% 可信区间。99% 的可信区间意味着如果变量不在这个范围之内, 这个变量的值出现的概率仅为 1%

错误注入实验也研究了对 Linux, Solaris 和 Windows 2000 系统的其他可用性策略, 如自动备份能力、重建速度和短暂错误等, 当然这些也没有相关文档说明。

对于短暂错误处理, 注入错误实验表明 Linux 的软件 RAID 实现采取了与 Solaris 和 Windows 相反的策略。Linux 的实现比较极端, 当第一次出错时系统将停止磁盘运转, 而不是去判断错误是否为短暂错误。相对来说 Solaris 和 Windows 的实现更宽容一些, 它们不考虑那些不再次发生的短暂错误。因此这些系统对于短暂错误体现出比 Linux 更好的灵活性。注意, 即使没有采取相应的措施, Windows 和 Solaris 也会记录短暂错误。当发生非短暂错误时, 这些系统的行为都是相似的。

6.5 排队论简介

在处理器设计中, 我们对与 CPI 公式有关的性能参数 (见第 1 章) 进行了简单的计算, 如果想得到更精确的数据, 可以采用更全面的系统仿真, 但这将花费更大的代价。在 I/O 系统中, 我们也

用一个理想化的案例来做了简单的计算,全面系统仿真当然会更精确,但是要得到预期的性能,计算量也将更大。

在 I/O 系统中,也需要一个数学工具来指导 I/O 设计。它比理想化的案例更精确,也有更大的计算量,但比系统仿真的工作量要小。基于 I/O 事件的可能性特征以及 I/O 资源的共享性,我们可以给出一系列的简单法则来计算整个 I/O 系统的响应时间和吞吐率。这部分的研究称为排队论 (queuing theory)。有许多书对这个问题都有介绍,本节只是对这个理论的一个简介。但即便如此,也能更好地指导 I/O 系统设计。

我们把 I/O 系统视为一个黑盒,如图 6.15 所示。在我们的例子中,处理器发出到达 I/O 设备的命令,当 I/O 设备完成时发出“离开”命令。



图 6.15 把 I/O 系统视为黑盒。我们通过这种方法发现一个简单却很重要的现象:如果系统处于稳定状态,那么进入系统的任务数量一定等于离开系统的任务数量。这种流平衡状态是稳定状态的必要条件,而非充分条件。系统只有经过足够长时间的测量或观察,而且平均等待时间稳定,才能说是达到了稳定状态

比起初始阶段的启动情况,我们更关心系统的长期或稳定状态。假设我们不关心这些,虽然有数学模型(如 Markov 链)的帮助,但是除了少数一些情况外,获得结果的唯一方法是仿真。本节的目的在于介绍比简单计算更复杂、比仿真更简单的方法。因而,我们在这里不会进行过多的分析(具体可参考附录 K)。

本节假设我们所评价的系统中多个独立 I/O 服务请求是均衡的,即输入速率等于输出速率。还假设不管每个任务等待服务的时间是多少,提供任务的速度都是固定的。在现实系统中任务执行速度实际取决于容量等系统特性,TPC-C 就是一个例子。

由此我们得到关于系统平均任务数、新任务的平均到达速率和任务平均执行时间的 Little 定律:

$$\text{系统平均任务数} = \text{到达速率} \times \text{平均响应时间}$$

只要黑盒内不产生新任务和撤销任务, Little 定律可以应用于任何均衡系统,注意到达速率和响应时间要使用同一时间单位,时间单位不一致是引起错误的普遍原因。

Little 定律也可以通过推导得到。假设我们观察一个系统 $\text{Time}_{\text{观察}}$ 分钟,在观察中,我们记录系统为每个任务服务的时间,然后汇总。在 $\text{Time}_{\text{观察}}$ 中完成的任务数为 $\text{Number}_{\text{任务}}$,整个系统汇总的时间总和为 $\text{Time}_{\text{总和}}$ 。注意任务在必要时会重叠,故 $\text{Time}_{\text{总和}} \geq \text{Time}_{\text{观察}}$ 。那么

$$\text{系统平均任务数} = \frac{\text{Time}_{\text{总和}}}{\text{Time}_{\text{观察}}}$$

$$\text{平均响应时间} = \frac{\text{Time}_{\text{总和}}}{\text{Number}_{\text{任务}}}$$

$$\text{到达速率} = \frac{\text{Number}_{\text{任务}}}{\text{Time}_{\text{观察}}}$$

第一个公式又可以写成

$$\frac{\text{Time}_{\text{总和}}}{\text{Time}_{\text{观察}}} = \frac{\text{Time}_{\text{总和}}}{\text{Number}_{\text{任务}}} \times \frac{\text{Number}_{\text{任务}}}{\text{Time}_{\text{观察}}}$$

如果我们将这三个定义代入上面的公式中，在右边交换结果，就能得到 Little 定律：

$$\text{系统平均任务数} = \text{到达速率} \times \text{平均响应时间}$$

我们将在稍后看到，这个简单的等式是非常有用的。

如果打开黑盒，将看到如图 6.16 所示内容。图中任务聚集并等待服务的区域称为队列或等待队列。执行所请求服务的设备称为服务器。直到本节的结束前两页，我们都假设是单服务器的情况。

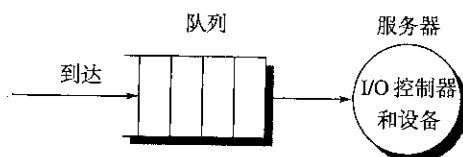


图 6.16 单服务器模型。这种情况下一个 I/O 请求在被服务器处理完毕之后就离开

由 Little 定律和一组定义可以推导出多个有用的等式：

- $\text{Time}_{\text{server}}$ ：每个任务的平均服务时间；平均服务速率 μ 为 $1/\text{Time}_{\text{server}}$ 。
- $\text{Time}_{\text{queue}}$ ：每个任务在队列中的平均等待时间。
- $\text{Time}_{\text{system}}$ ：每个任务在系统中的平均时间或称响应时间，它为 $\text{Time}_{\text{server}}$ 与 $\text{Time}_{\text{queue}}$ 之和。
- 到达速率：每秒平均到达的任务数。用 λ 来表示。
- $\text{Length}_{\text{server}}$ ：服务中的平均任务数。
- $\text{Length}_{\text{queue}}$ ：平均队列长度。
- $\text{Length}_{\text{system}}$ ：系统中的平均任务数，它为 $\text{Length}_{\text{server}}$ 与 $\text{Length}_{\text{queue}}$ 之和。

在服务开始之前，一个任务要在队列中的等待时间 ($\text{Time}_{\text{queue}}$) 与一个任务在完成前需要等待的时间 ($\text{Time}_{\text{system}}$) 是两个使人易产生误解的概念。后者就是所谓的响应时间，二者之间的关系是 $\text{Time}_{\text{system}} = \text{Time}_{\text{queue}} + \text{Time}_{\text{server}}$ 。

Little 定律也可以表示为：平均服务中的任务数 ($\text{Length}_{\text{server}}$) = 到达速率 \times $\text{Time}_{\text{server}}$ 。服务器利用率 = 平均服务中的任务数 / 服务率，对于单个服务器，服务率为 $1/\text{Time}_{\text{server}}$ 。服务器利用率（在这种情况下为单一服务器的平均任务数）可以简单地表示为

$$\text{服务器利用率} = \text{到达速率} \times \text{Time}_{\text{server}}$$

其值一定在 0 和 1 之间，否则就是发生了到达的任务数多于服务器所能提供服务的任务数的情况，这不符合系统均衡性假设。注意，上述公式只是 Little 定律的另一种表示形式。利用率也称为流量强度 (traffic intensity)，一般用符号 ρ 表示。

例题 假设一个 I/O 系统只有 1 个磁盘，每秒可接收 50 个 I/O 请求，磁盘对每个 I/O 请求的平均服务时间是 10 ms，求 I/O 系统的利用率。

解答：利用上面的等式，把 10 ms 表示为 0.01 s，可得

$$\text{服务器利用率} = \text{到达速率} \times \text{Time}_{\text{server}} = 50/\text{s} \times 0.01 \text{ s} = 0.50$$

因此，该 I/O 系统服务器的利用率为 0.5。

队列如何提交任务给服务器称为队列规则,最简单也是最常用的规则是先进先出(FIFO),如果使用先进先出规则,我们可以将任务在队列中的等待时间与队列中的平均任务数联系在一起:

$$\text{Time}_{\text{queue}} = \text{Length}_{\text{queue}} \times \text{Time}_{\text{server}} + \text{新任务到达时正在接受服务的任务的平均完成时间}$$

也就是说,排队时间等于队列中任务数乘以平均服务时间,加上一个新任务到达时服务器完成正在接受服务的任务所需的时间(后面的讨论中对任务到达有更严格的定义)。

等式最后一项比前面的项要复杂一些。一个新的任务可能在任何时刻到达,因此,我们无法得知一个任务会在服务器中停留多长时间。虽然这些请求都是随机事件,但如果我们知道事件分布的规律,就可以对性能做出预测。

随机变量的泊松分布

为对公式的最后一项做出评估,需要对随机变量分布情况有所了解。如果一个变量在一个指定集合内按一定的概率取值,那么该变量就是随机的;也就是说,我们虽然不知道下一个变量究竟取哪一个值,但我们知道变量所有可能取值的概率。

因为操作系统通常调度多个独立的I/O请求进程,所以I/O系统服务请求可看做是随机变量模型。我们也可以根据寻道和旋转时延等性质决定随机变量,构建I/O服务时间模型。

一种表示随机变量取值分布的方法是柱状图,它将最小值到最大值间的区域划分成一个个子域,并称之为桶。柱状图以长条竖状块的形式表示桶中的数。

柱状图特别适合于对离散数值分布的描述——例如I/O请求数。对于那些非离散数值,如等待一个I/O请求的时间,我们有两种选择:(1)可以用一条曲线来表示所有取值范围内的值,便于我们准确地对变量值进行估计;(2)使用一个合适的时间单位,这样就可以得到非常多的桶来精确地估计时间。例如,尽管磁盘服务时间是连续的,我们仍可以用10 μs为时间间隔来建立磁盘服务时间柱状图。

因此,对于上面等式的最后一项,可以用平均时间和方差来描述这个随机变量分布的特性。前者可用加权算术平均时间来描述。假定先测定任务发生次数,比如说是 n_i ,由此可以计算任务 i 发生的频率:

$$f_i = \frac{n_i}{\left(\sum_{i=1}^n n_i \right)}$$

加权算术平均为

$$\text{加权算术平均时间} = f_1 \times T_1 + f_2 \times T_2 + \cdots + f_n \times T_n$$

T_i 是任务 i 所需时间, f_i 是任务 i 发生的频率。

人们通常使用标准偏差来描述一组数的偏离程度,这里我们用方差来描述,即标准差的平方。给定加权算术平均值,其方差可由下式计算:

$$\text{方差} = (f_1 \times T_1^2 + f_2 \times T_2^2 + \cdots + f_n \times T_n^2) - \text{加权算术平均时间}^2$$

计算方差时所用的单位很重要。假设是时间分布,如果时间以100 ms为单位计算,平方后则得到一个10 000 ms²的方差,这个单位显然并不通用。如果能不采用单位进行计算,那会方便很多。

采用协方差方法可以避免由于单位带来的问题,协方差一般用 C^2 表示:

$$C^2 = \frac{\text{方差}}{\text{加权算术平均时间}^2}$$

我们可以得到方差系数 C :

$$C = \frac{\sqrt{\text{方差}}}{\text{算术平均时间}} = \frac{\text{标准差}}{\text{算术平均时间}}$$

我们试图得出随机事件一些特性,但为了预测性能,我们需要描述随机事件的分布,这样才可以采用数学分析工具进行分析。最常用的随机分布为指数分布,这里 C 值为 1。

请注意,我们这里使用的是常系数 C 去描述平均值的变化特征。由于 C 值不随时间变化而变化,所以历史事件不会影响现在事件发生的可能性。这种性质称为无记忆性,它是使用这些模型预测行为的关键假设(如果不存在这种失忆,那么我们就不得不关心每个请求相对彼此的到达时间,这就大大削弱了使用数学模型的意义)。

泊松分布(Poisson distribution)是一个最为常用的指数分布,它以数学家西蒙·泊松命名。它用以描述既定时间间隔内随机事件的特征,具有许多有用的数学性质。泊松分布可写成如下等式(称为概率群分布函数):

$$\text{Probability}(k) = \frac{e^{-a} \times a^k}{k!}$$

其中 a = 事件速率 \times 占用时间。如果事件到达的间隔时间是指数分布的且用到达速率代替事件速率,那么在 t 时间间隔内到达的数量是一个泊松过程,符合泊松分布,其中 a = 到达速率 $\times t$ 。在上页曾提到, $\text{Time}_{\text{server}}$ 等式中对任务到达有另一限制,那就是它必须符合泊松过程。

最后,我们可以计算一个新任务等待服务器完成一个正在执行的任务所需的时间(这里仍然假设任务到达符合泊松分布),这一时间称为平均剩余服务时间(average residual service time):

$$\text{平均剩余服务时间} = 1/2 \times \text{加权平均时间} \times (1 + C^2)$$

这里我们不会对这一公式做任何推导,但是可以完全相信直觉的判断。当事件分布不是随机的并且所有可能取值等于平均值时,标准差为 0,因此 C 也为 0。正如我们所预料的,平均剩余服务时间只是平均服务时间的一半。如果事件到达是随机的并且服从泊松分布,那么 C 为 1,平均剩余服务时间等于加权算术平均时间。

例题 使用以上定义和公式,根据平均服务时间($\text{Time}_{\text{server}}$)和服务利用率推导任务在队列中的平均等待时间($\text{Time}_{\text{queue}}$)。

解答: 队列中的所有任务($\text{Length}_{\text{queue}}$)都要在新任务得到服务前完成;每个任务的平均服务时间都为 $\text{Time}_{\text{server}}$;一个正在服务中的任务需要平均剩余服务时间完成。服务器处于忙状态的概率则为服务器利用率。因此,期望服务时间为服务器利用率 \times 平均剩余服务时间,由此导出我们最初的公式:

$$\text{Time}_{\text{queue}} = \text{Length}_{\text{queue}} \times \text{Time}_{\text{server}} + \text{服务器利用率} \times \text{平均剩余服务时间}$$

将平均剩余服务时间的定义导入上面的公式,再用到达速率 $\times \text{Time}_{\text{queue}}$ 替换 $\text{Length}_{\text{queue}}$ 可以得到

$$\text{Time}_{\text{queue}} = \text{服务器利用率} \times (1/2 \times \text{Time}_{\text{server}} \times (1 + C^2)) + (\text{到达速率} \times \text{Time}_{\text{queue}}) \times \text{Time}_{\text{server}}$$

本节只考虑指数分布的情况, C^2 为1,因此

$$\text{Time}_{\text{queue}} = \text{服务器利用率} \times \text{Time}_{\text{server}} + (\text{到达速率} \times \text{Time}_{\text{queue}}) \times \text{Time}_{\text{server}}$$

用服务器利用率代替到达速率 $\times \text{Time}_{\text{server}}$ 得

$$\begin{aligned} \text{Time}_{\text{queue}} &= \text{服务器利用率} \times \text{Time}_{\text{server}} + (\text{到达速率} \times \text{Time}_{\text{server}}) \times \text{Time}_{\text{queue}} \\ &= \text{服务器利用率} \times \text{Time}_{\text{server}} + \text{服务器利用率} \times \text{Time}_{\text{queue}} \end{aligned}$$

经简化得

$$\begin{aligned} \text{Time}_{\text{queue}} &= \text{服务器利用率} \times \text{Time}_{\text{server}} + \text{服务器利用率} \times \text{Time}_{\text{queue}} \\ \text{Time}_{\text{queue}} - \text{服务器利用率} \times \text{Time}_{\text{queue}} &= \text{服务器利用率} \times \text{Time}_{\text{server}} \\ \text{Time}_{\text{queue}} \times (1 - \text{服务器利用率}) &= \text{服务器利用率} \times \text{Time}_{\text{server}} \\ \text{Time}_{\text{queue}} &= \text{Time}_{\text{server}} \times \frac{\text{服务器利用率}}{(1 - \text{服务器利用率})} \end{aligned}$$

Little 定律也适用于黑盒组件,因为这些组件也满足平衡性:

$$\text{Length}_{\text{queue}} = \text{到达速率} \times \text{Time}_{\text{queue}}$$

如果将上面所得代入 $\text{Time}_{\text{queue}}$,则得到

$$\text{Length}_{\text{queue}} = \text{到达速率} \times \text{Time}_{\text{server}} \times \frac{\text{服务器利用率}}{(1 - \text{服务器利用率})}$$

因为到达速率 $\times \text{Time}_{\text{server}} = \text{服务器利用率}$,所以可以把公式进一步简化:

$$\text{Length}_{\text{queue}} = \text{服务器利用率} \times \frac{\text{服务器利用率}}{(1 - \text{服务器利用率})} = \frac{\text{服务器利用率}^2}{(1 - \text{服务器利用率})}$$

例题 对于265页中的第一个例子的系统,服务器利用率为0.5,I/O请求队列的平均长度是多少?

解答: 用上述等式有

$$\text{Length}_{\text{queue}} = \frac{\text{服务器利用率}}{(1 - \text{服务器利用率})} = \frac{0.5^2}{(1 - 0.5)} = \frac{0.25}{0.50} = 0.5$$

因此上述队列平均有0.5个要求。

如前所述,这些等式基于排队论的数学模型,即用等式预测随机事件的行为。现实系统对于排队论来说太复杂,以至于很难给出精确的分析,因此,只有需要近似结论时排队论才能发挥作用。

排队论对用简单算术描述过去事件和需用复杂数学分析的未来事件做了明确的区分。在计算机系统中,我们通常用过去事件来预测未来事件;如LRU块替换算法(见第5章)就是这样。因

此,这里对过去事件的评价和对未来事件的预测分布之间的区分并不是那么明显。我们使用对过去事件的评价来验证分布的类型,并在后面会用到这一分布的特性。

我们再回顾一下关于队列模型的假设:

- 系统是均衡的。
- 两个相继到达的请求间隔称为**到达间隔时间**,它是按指数分布的。
- 请求的数目是没有限制的[在排队论中称为**无限总体模型**,有限总体模型用于到达速率随系统已有任务数变化的模型]。
- 服务器在完成对前面一个客户的服务之后立即开始对下一个客户的服务。
- 队列长度没有限制,遵循先进先出原则,因此队列中的所有任务都会被完成。
- 只有一个服务器。

这样的队列称为 **M/M/1 队列**: M 表示按指数分布的随机请求到达 ($C^2 = 1$), 其中 M 代表 A. A. Markov, 他是定义和分析无记忆性质的数学家; M 表示按指数分布的随机服务时间 ($C^2 = 1$), M 仍代表 Markov; 1 表示一个服务器。

M/M/1 模型是一个简单而广泛使用的模型。

通常在队列示例中使用指数分布有三个原因,一个好,一个不好,还有一个是不好不坏。好的原因是许多任意分布的集合呈现为指数分布。很多情况下,计算机系统上的某一特殊行为都是许多部件交互作用的结果,因此,到达间隔时间的指数分布是正确的模型。不好不坏的原因是当变量模糊时,用 $C = 1$ 的指数分布比选择低偏离度 ($C \approx 0$) 或选择高偏离度 (C 很大) 更安全。不利的原因是如果假设为指数分布,则数学描述过于简单。

下面把排队论应用到几个实际例子中。

例题 假设一个处理器每秒发出 40 个磁盘 I/O 请求,这些请求按指数分布,平均磁盘服务时间为 20 ms,回答下列问题。

1. 该磁盘的平均利用率是多少?
2. 用于排队的平均时间是多少?
3. 磁盘请求的平均响应时间是多少,包括排队时间和磁盘服务时间。

解答: 我们重申一下如下事实:

平均每秒到达任务数为 40。

对一个任务服务的平均磁盘时间为 20 ms (0.02 s)。

则服务器利用率为

$$\text{服务器利用率} = \text{到达速率} \times \text{Time}_{\text{server}} = 40 \times 0.02 = 0.8$$

由于服务时间呈指数分布,我们可以用简化的公式计算平均队列等待时间:

$$\begin{aligned} \text{Time}_{\text{queue}} &= \text{Time}_{\text{server}} \times \frac{\text{服务器利用率}}{(1 - \text{服务器利用率})} \\ &= 20 \text{ ms} \times \frac{0.8}{1 - 0.8} = 20 \times \frac{0.8}{0.2} = 20 \times 4 = 80 \text{ ms} \end{aligned}$$

因此平均响应时间为

$$\text{Time}_{\text{system}} = \text{Time}_{\text{queue}} + \text{Time}_{\text{server}} = 80 + 20 = 100 \text{ ms}$$

因此, 80% 的时间用于在队列中等待!

例题 假设有一个新的更快的磁盘, 重新回答上面的问题, 假设磁盘服务时间仅为 10 ms。

解答: 磁盘利用率为

$$\text{服务器利用率} = \text{到达速率} \times \text{Time}_{\text{server}} = 40 \times 0.01 = 0.4$$

利用公式计算平均队列等待时间:

$$\begin{aligned} \text{Time}_{\text{queue}} &= \text{Time}_{\text{server}} \times \frac{\text{服务器利用率}}{1 - \text{服务器利用率}} \\ &= 10 \text{ ms} \times \frac{0.4}{1 - 0.4} = 10 \times \frac{0.4}{0.6} = 10 \times \frac{2}{3} = 6.7 \text{ ms} \end{aligned}$$

平均响应时间为 $10 + 6.7 = 16.7 \text{ ms}$, 虽然新的服务时间比原来只快了 2.0 倍, 但平均响应时间比原来快了 6.0 倍。

上面我们假设是单一服务器, 例如具有单一磁盘的系统。但在实际系统中通常有多个磁盘, 因此可能使用多台服务器。这种系统称为排队理论的 M/M/m 模型, 如图 6.17 所示。

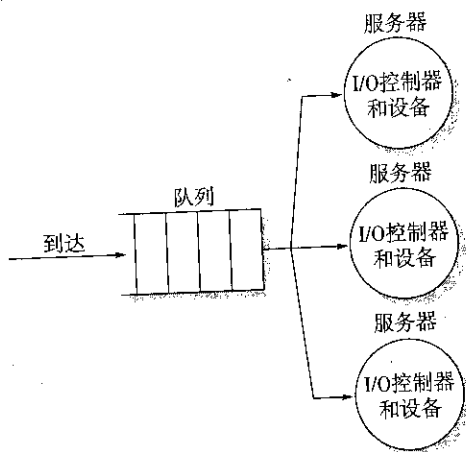


图 6.17 M/M/m 多服务器模型

下面我们给出 M/M/m 队列的公式, 用 N_{servers} 表示服务器数量。那么由前面两个公式很容易就可以得到

$$\text{利用率} = \frac{\text{到达速率} \times \text{Time}_{\text{server}}}{N_{\text{servers}}}$$

$$\text{Length}_{\text{queue}} = \text{到达速率} \times \text{Time}_{\text{queue}}$$

在队列中的等待时间为

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{P_{\text{tasks} \geq N_{\text{servers}}}}{N_{\text{servers}} \times (1 - \text{利用率})}$$

这个公式与 M/M/1 中的公式相似,只是用一个任务在队列中等待的概率(即它将不再立即被执行)代替单服务器的利用率,然后再除以服务器的个数。在具有多个服务器的系统中计算任务在队列中等待的概率很复杂。首先,系统中无任务的概率为

$$\text{Prob}_{0 \text{ tasks}} = \left[1 + \frac{(N_{\text{servers}} \times \text{利用率})^{N_{\text{servers}}}}{N_{\text{servers}}! \times (1 - \text{利用率})} + \sum_{n=1}^{N_{\text{servers}}-1} \frac{(N_{\text{servers}} \times \text{利用率})^n}{n!} \right]^{-1}$$

其次,系统中有与服务器数目一样多或者更多任务时的概率为

$$\text{Prob}_{\text{tasks} \geq N_{\text{servers}}} = \frac{N_{\text{servers}} \times \text{利用率}^{N_{\text{servers}}}}{N_{\text{servers}}! \times (1 - \text{利用率})} \times \text{Prob}_{0 \text{ tasks}}$$

注意,如果 N_{servers} 为 1, $\text{Prob}_{\text{tasks} \geq N_{\text{servers}}}$ 化简为利用率,得到的模型与 M/M/1 模型相同。我们来举个例子。

例题 假设除了一个新的速度较快的磁盘外,我们又添加了另一个较慢的磁盘来复制数据,读数据可以利用其中任何一个盘。假设所有请求都是读,用 M/M/m 队列来计算上面的问题。

解答: 两个磁盘的平均利用率为

$$\text{服务器利用率} = \frac{\text{到达速率} \times \text{Time}_{\text{server}}}{N_{\text{servers}}} = \frac{40 \times 0.02}{2} = 0.4$$

我们首先计算队列中无任务的概率:

$$\begin{aligned} \text{Prob}_{0 \text{ tasks}} &= \left[1 + \frac{(2 \times \text{利用率})^2}{2! \times (1 - \text{利用率})} + \sum_{n=1}^1 \frac{(2 \times \text{利用率})^n}{n!} \right]^{-1} \\ &= \left[1 + \frac{(2 \times 0.4)^2}{2 \times (1 - 0.4)} + (2 \times 0.4) \right]^{-1} = \left[1 + \frac{0.640}{1.2} + 0.800 \right]^{-1} \\ &= [1 + 0.533 + 0.800]^{-1} = 2.333^{-1} \end{aligned}$$

我们用这个结果计算队列中有任务的概率:

$$\begin{aligned} \text{Prob}_{\text{tasks} \geq N_{\text{servers}}} &= \frac{2 \times \text{利用率}^2}{2! \times (1 - \text{利用率})} \times \text{Prob}_{0 \text{ tasks}} \\ &= \frac{(2 \times 0.4)^2}{2 \times (1 - 0.4)} \times 2.333^{-1} = \frac{0.640}{1.2} \times 2.333^{-1} \\ &= 0.533 / 2.333 = 0.229 \end{aligned}$$

最后,计算在队列中的等待时间:

$$\begin{aligned} \text{Time}_{\text{queue}} &= \text{Time}_{\text{server}} \times \frac{\text{Prob}_{\text{tasks} \geq N_{\text{servers}}}}{N_{\text{servers}} \times (1 - \text{利用率})} \\ &= 0.020 \times \frac{0.229}{2 \times (1 - 0.4)} = 0.020 \times \frac{0.229}{1.2} \\ &= 0.020 \times 0.190 = 0.0038 \end{aligned}$$

平均响应时间为 $20 + 3.8 = 23.8 \text{ ms}$ 。对于这个工作量,两磁盘系统可以将队列等待时间减少为使用单个慢速磁盘系统的 $1/21$ 、使用单个快速磁盘系统的 $1/1.75$ 。使用单个快速磁盘的系统平均服务时间比两磁盘系统快 1.4 倍,因为磁盘服务时间比两磁盘系统快 2.0 倍。

如果我们在多队列和多服务器上推广 M/M/m 模型将会非常好，因为这一步更加现实。但此模型较难实现和使用，故在这里我们不涉及。

6.6 相关问题

点到点连接和用交换机代替总线

在摩尔定律的持续作用下，各部件的成本不断降低，点到点连接和交换机应用越来越广泛。综合考虑来自快速处理器、快速磁盘和快速局域网对更高 I/O 带宽的要求，总线成本优势的下降意味着总线在桌面和服务器计算机中的竞争力日益减弱。这个趋势已在高性能服务器中显现，本书的最后一章会提到这一点。在 2006 年，该趋势又全面扩展到存储系统中。图 6.18 给出了老式总线标准及其升级产品。

标准	数据位宽	线长 (m)	时钟频率	每分钟的 数据率 (MB/s)	最大 I/O 设备数
并行 ATA	8	0.5	133 MHz	133	2
串行 ATA	2	2	3 GHz	300	?
SCSI	16	12	80 MHz	320	15
Serial Attach SCSI	1	10	(DDR)	375	16 256
PCI	32/64	0.5	33/66 MHz	533	?
PCI Express	2	0.5	3 GHz	250	?

图 6.18 并行 I/O 总线和其点到点的交换。注意到位数和带宽是按方向传输的，故双向传输时，带宽加倍

下一代技术的位数和带宽是按每个方向定义的，因此双向性能指标会翻倍。因为新的设计思想使用了较少的线，一种通用的方法是增加带宽来提供位数和带宽的多次描述。

块服务器与文件管理器的对比

到目前为止，我们在很大程度上忽略了操作系统在存储中所扮演的角色。和编译器使用指令系统相类似的方法，操作系统根据硬件的实际使用情况来决定执行何种 I/O 技术。操作系统通常将文件抽象保存在磁盘一开始的磁盘块上。逻辑单位、逻辑卷和物理卷这些相关术语在 Microsoft 和 UNIX 系统中是指磁盘块集合的子集。

一个逻辑单元是从磁盘阵列输出的基本存储单元，通常从磁盘阵列的子集中创建。逻辑单元对服务器来说是一个单独的虚拟“磁盘”。在 RAID 磁盘阵列中，逻辑单元被设计成特殊的 RAID 布局，如 RAID 5。物理卷是文件系统用来访问逻辑单元的设备文件。逻辑卷提供了一个虚拟化层面使文件系统可以在多个盘片上分割物理卷或在多个物理卷上带状分布数据。逻辑单元是磁盘阵列的抽象，对于操作系统它代表虚拟的磁盘。而物理卷和逻辑卷是操作系统所使用的抽象概念，用来把这些虚拟磁盘分割成更小的独立文件系统。

上面已经涉及了一些块集合的术语。然后问题出现了，文件映像应该在哪里维护：是在服务器中还是存储区域网络的另一端？

传统的答案是服务器。它访问存储器并维护元数据。大多数文件系统使用文件高速缓存，因此，服务器必须保持文件访问的一致性。虽然磁盘可以直接访问——在服务器连接的 I/O 总线上定位——或者通过存储区域网络 (SAN) 访问，但事实上是由服务器传送数据块到存储子系统的。

另一个答案是磁盘子系统自己维护文件抽象,由服务器使用文件系统协议与存储器通信。例如,UNIX系统中的网络文件系统(NFS)和Windows系统中的通用互联网文件系统(CIFS)都是这样的文件系统协议。这些设备称为网络附加存储设备(NAS),因为把这些存储器直接连到服务器是毫无意义的。但是,这个名称也有一些错误,因为像FC-AL这样的存储网络可以用于连接块服务器。文件管理器这一术语经常用于仅提供文件服务和文件存储的NAS设备。Network Appliances是第一家生产文件管理器的公司。

人们将存储器用于网络主要是为了方便众多电脑之间对数据的共享以及使操作人员维护更容易。

异步 I/O 和操作系统

磁盘一般在机械延迟上花费的时间比传输数据的时间更多。因而,使一个程序并行地访问多个磁盘来获取数据,以提高 I/O 性能是一种自然的选择。

对于 I/O 而言,最直接的策略是请求数据并使用。操作系统随后会切换到另一个进程,直到期望的数据到达后再切换回原进程。这种方法称为同步 I/O——进程等待数据从磁盘中读出。

另一种可供选择的策略是进程在请求后继续运行,直到它试图读取请求数据时才阻塞。这样的异步 I/O 允许进程继续发出请求,因此,许多 I/O 请求可以同时操作。异步 I/O 像乱序执行 CPU 中的 Cache 一样使用同一种策略,即通过复合处理事务获得更大的带宽。

6.7 I/O 系统设计与评价——互联网存储档案集群

I/O 系统设计就是要找出一种设计方案,满足成本、可靠性和设备多样性等要求,同时避免造成 I/O 性能的瓶颈。这就需要在存储器和 I/O 设备间进行折中,因为性能——以及由此产生的有效性价比——是与 I/O 链中性能最差的链接直接相关的。系统结构设计者同时也需要考虑系统的可扩展性,以便使用户根据应用对 I/O 进行定制。这种可扩展性(包括 I/O 设备的数量和类型)因为需要更长的 I/O 总线、更大的电源以支持 I/O 设备和更大的机箱,所以需要一定的开销。

在设计 I/O 系统时,需根据不同的 I/O 连接模式和每种 I/O 设备不同数量分析系统的性能、成本、能力和可用性。这里介绍 I/O 系统设计的一系列步骤,其中每一步都是以市场需求或简单的成本、性能和可用性为目标的。

1. 列出连接到计算机上的各种 I/O 设备类型,或列出计算机所支持的标准总线。
2. 列出每种 I/O 设备的物理需求,包括容量、功率、连接插口、总线槽和扩展槽位等。
3. 列出每种 I/O 设备的成本,包括其控制器的成本。
4. 列出每个 I/O 设备的可靠性。
5. 记录每种 I/O 设备所需的处理器资源,包括:

- 初始化 I/O、支持 I/O 设备操作(如处理中断)和完成 I/O 所需要的指令周期数。
- 由于 I/O 使用主存、总线或 Cache 而引起的处理器停顿时钟周期数。
- 处理器用于从 I/O 活动恢复到正常计算(如 Cache 刷新)所需的时钟周期数。

6. 列出每种 I/O 设备的存储器和 I/O 总线资源需求。即使处理器没有使用存储器,存储器和 I/O 总线带宽也是受限的。

7. 最后一步是对这些 I/O 设备的各种组织方式进行性能和可用性评价。性能虽然可以用排队论模型进行评价, 但通过模拟的方法可获得比较准确的计算。假设 I/O 设备的故障是独立的, 且失效时间呈指数分布, 则可计算可靠性。可用性可由可靠性计算得出, 但是需要把从故障出现到修复的时间都计算在内, 并估计设备的 MTTF。

这样, 就可以根据成本、性能和可用性等目标选择最好的组织方案。

性价比会影响 I/O 模式的选择和物理设计。根据不同应用, 性能可以用每秒传输多少 MB 或完成多少次 I/O 操作来表示。实现高性能的受限因素有 I/O 设备的速度、数量以及主存和处理器的速度。影响低成本的因素主要是 I/O 设备本身。可用性实现部分取决于组织方法的不可用性的代价。

接下来, 我们会跳过设计这一环, 直接对一个实际的系统进行评价。

互联网存储档案集群

为了能更清楚地说明这些设计思想, 我们将评价一款用于互联网档案面向大规模存储的集群, 包括其成本、性能和可用性。互联网档案开始于 1996 年, 其作用是随时间的改变记录互联网上的历史记录。可以使用时光倒流机器 (Wayback Machine) 到互联网档案的接口, 访问一个 Web 站点过去某个时间的 URL。在 2006 年, 此档案超过 1 PB (10^{15} 字节) 容量, 并且还在以每月 12 TB (10^{12} 字节) 新数据的速度增长, 故可扩充的存储系统是必要的。除了存储历史记录之外, 硬件还需要每隔几个月从互联网上收集 Web 快照。

由局域网互连的计算集群是一种很经济的计算方式, 它能很好地适应某些应用。集群也在互联网服务中扮演很重要的角色, 例如 Google 的搜索引擎, 不过它更关注存储而非计算。

虽然很多年来互联网档案使用了多种硬件, 但目前它正发展为一种新的集群, 这在功耗和体积上都是有一定的优势。其最基本的组成模块是一个 1U 的存储节点, 称为 PetaBox GB2000, 使用 Capricorn 技术。在 2006 年, 它包含了 4 个 500 GB 的并行 ATA 磁盘、512 MB 的 DDR266 DRAM 存储器、一个 10/100/1000 的以太网接口和一个 VIA 的 1 GHz C3 处理器, 使用 80x86 指令系统。此节点在通用的配置下, 功耗为 80 W。

图 6.19 给出了集群在标准 VME 机柜上的情况。标准的 VME 机柜能装满 40 个 GB2000, 提供了 80 TB 的原始容量。40 个节点由一个 48 端口的 10/100 或 10/100/1000 交换机互连, 消耗 3 kW 的功耗。每个机柜的极限功耗是 10 kW, 故能满足需求。

1 PB 需要 12 个这样的机柜, 由 Gbit 的高端交换机将每个机柜连接起来。

互联网存储档案集群的成本、性能和可靠性评价

为举例说明如何评价一个 I/O 系统, 我们需要估计一下此集群的成本、性能和组件的可靠性。我们对其成本和性能做以下假设:

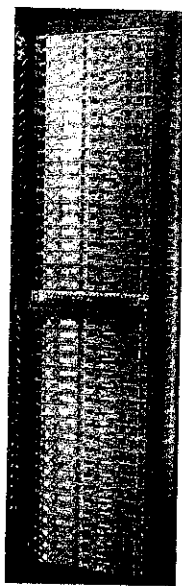


图 6.19 互联网档案所使用的 Capricorn 系统的 TB-80 VME 机柜。所有电缆、交换机和显示器可从前端访问, 后端仅仅用来通风。因此可以把两个机柜背对背放, 这样可以减少机房的占地空间

- 一个VIA处理器、512 MB的DDR266存储器、ATA磁盘控制器、电源、风扇以及机箱的成本是500美元。
- 每四个7200 rpm的并行ATA硬盘组成500 GB, 平均寻道时间为8.5 ms, 磁盘的数据传输率为50 MB/min, 成本为375美元。PATA连接的速度是133 MB/min。
- 48端口的10/100/1000以太网交换机和一个机柜所有电缆的成本为3000美元。
- VIA处理器的性能是1000 MIPS。
- ATA控制器处理每个磁盘I/O操作需增加0.1 ms的开销。
- 操作系统处理每个磁盘I/O操作需要50 000个CPU指令。
- 网络协议栈需要100 000个CPU指令在集群和外界间传输一个数据块。
- 使用Wayback接口访问一个历史数据的平均I/O大小为16 KB, 当收集一个新的快照时为50 KB。

例题 评价80 TB机柜每秒每个I/O操作(IOPS)的成本。假设每次磁盘I/O操作需要的寻道时间和旋转延迟均为平均值。假设所有设备的能力都可以得到100%的发挥, 并且所有磁盘负载是均匀分布的。换言之, 当系统仅仅取决于链接能力最弱的部件, 且该部件能100%利用。分别计算平均I/O大小。

解答: I/O性能是受限于I/O链中链接能力最弱的部件的, 因此我们通过对每种组织方式的每个链接的性能进行评价, 来确定对应组织方式的性能。

首先计算一个GB2000的CPU、存储器和I/O总线的最大IOPS数量。CPU的I/O性能是由CPU的速度和执行一次磁盘I/O操作所需的指令数决定的:

$$\text{CPU的最大IOPS数量} = \frac{1000 \text{ MIPS}}{50\,000 \text{ 指令每次I/O} + 100\,000 \text{ 指令每次消息}} = 6667 \text{ IOPS}$$

存储系统的最大性能是由存储器带宽和每次I/O传输数据决定的:

$$\text{主存的最大IOPS数量} = \frac{266 \times 8}{16 \text{ KB 每次I/O}} \approx 133\,000 \text{ IOPS}$$

$$\text{主存的最大IOPS数量} = \frac{266 \times 8}{50 \text{ KB 每次I/O}} \approx 42\,500 \text{ IOPS}$$

并行ATA连接的性能受限于带宽和每次I/O传输数据量:

$$\text{I/O总线的最大IOPS数量} = \frac{133 \text{ MB/s}}{16 \text{ KB 每次I/O}} \approx 8300 \text{ IOPS}$$

$$\text{I/O总线的最大IOPS数量} = \frac{133 \text{ MB/s}}{50 \text{ KB 每次I/O}} \approx 2700 \text{ IOPS}$$

由于有两种总线, I/O总线限制了最大的性能: 对于16 KB的块是8300 IOPS, 对于50 KB的块是2700 IOPS。

现在, 我们再来看看I/O链上下一个链接(ATA控制器)的性能。在PATA通道上传输一个数据块的时间是

$$\text{并行ATA的传输时间} = \frac{16 \text{ KB}}{133 \text{ MB/s}} = 0.1 \text{ ms}$$

$$\text{并行ATA的传输时间} = \frac{50 \text{ KB}}{133 \text{ MB/s}} = 0.4 \text{ ms}$$

ATA 控制器开销，意味着每个 I/O 花费 0.2~0.5 ms，每个磁盘控制器最大的速

$$\text{每个 ATA 控制器的最大 IOPS} = \frac{1}{0.2 \text{ ms}} = 5000 \text{ IOPS}$$

$$\text{每个 ATA 控制器的最大 IOPS} = \frac{1}{0.5 \text{ ms}} = 2000 \text{ IOPS}$$

接下来 I/O 链是磁盘自身。平均磁盘 I/O 时间是

$$\text{I/O 时间} = 8.5 \text{ ms} + \frac{0.5}{7200 \text{ RPM}} + \frac{16 \text{ KB}}{50 \text{ MB/s}} = 8.5 + 4.2 + 0.3 = 13.0 \text{ ms}$$

$$\text{I/O 时间} = 8.5 \text{ ms} + \frac{0.5}{7200 \text{ RPM}} + \frac{50 \text{ KB}}{50 \text{ MB/s}} = 8.5 + 4.2 + 1.0 = 13.7 \text{ ms}$$

因此，磁盘性能为

$$\text{每个磁盘的最大 IOPS (使用平均寻道)} = \frac{1}{13.0 \text{ ms}} \approx 77 \text{ IOPS}$$

$$\text{每个磁盘的最大 IOPS (使用平均寻道)} = \frac{1}{13.7 \text{ ms}} \approx 73 \text{ IOPS}$$

即对于 4 个磁盘来说是 292~308 IOPS。

I/O 链中最后的部分是在外层互联计算机的网络。连接的速度决定了限制的大小：

$$\text{每 1000 Mbit 以太网连接的最大 IOPS} = \frac{1000 \text{ Mbit}}{16\text{K} \times 8} = 7812 \text{ IOPS}$$

$$\text{每 1000 Mbit 以太网连接的最大 IOPS} = \frac{1000 \text{ Mbit}}{50\text{K} \times 8} = 2500 \text{ IOPS}$$

很显然，GB2000 的性能瓶颈是磁盘。整个系统的 IOPS 是 $40 \times 308 = 12\,320 \text{ IOPS}$ 或 $40 \times 292 = 11\,680 \text{ IOPS}$ 。网络交换机如果不能支持 $12\,320 \times 16\text{K} \times 8 = 1.6 \text{ Gb/s}$ 或 $11\,680 \times 50\text{K} \times 8 = 4.7 \text{ Gb/s}$ ，则也会成为瓶颈。我们假设 48 口的交换机上另外 8 Gbit 的端口将集群和外界相连，故它能支持集群中全部 160 个磁盘全速的 IOPS。

根据这些假设，一个 80 TB 的机柜的成本为 $40 \times (\$500 + 4 \times \$375) + \$3000 + \$1500 = \$84\,500$ 。磁盘占了全部成本的 60%。每 TB 的成本大约为 \$1000，这比先前 2001 年时的存储集群提高了 10~15 倍。每个 IOPS 的成本为 \$7。

计算 TB-80 集群的 MTTF

类似 Google 的互联网服务通过应用层的多副本来提供可靠性，通常分布在不同的地理站点，以防范环境因素和硬件故障造成的损坏。因此，互联网存储档案在每个站点有两个数据备份，这些站点分别在旧金山和阿姆斯特丹，以及埃及的亚历山大。每个站点存有较高价值内容的双重备份（如音乐、书籍、电影和视频）和 Web 历史纪录的单一备份。为降低成本，80-TB 集群中没有冗余。

例题 我们来看看平均故障时间。比起使用制造商提供的 600 000 小时的 MTTF，我们将使用最近硬盘研究综述中的数据[Gray 和 van Ingen 2005]。正如第 1 章提到的，大约 3%~7% 的 ATA 硬盘每年都有故障，MTTF 大约为 125 000~300 000 小时。做以下假设，并假设寿命按指数分布：

- CPU/主存/机箱的 MTTF 为 1 000 000 小时。
- PATA 磁盘的 MTTF 为 125 000 小时。
- PATA 磁盘控制器的 MTTF 为 500 000 小时。
- 以太网交换机的 MTTF 为 500 000 小时。
- 电源的 MTTF 为 200 000 小时。
- 风扇的 MTTF 为 200 000 小时。
- PATA 电缆的 MTTF 为 1 000 000 小时。

解答：根据以上数据，我们计算故障率为

$$\begin{aligned}\text{故障率} &= \frac{40}{1\,000\,000} + \frac{160}{125\,000} + \frac{40}{500\,000} + \frac{1}{500\,000} + \frac{40}{200\,000} + \frac{40}{200\,000} + \frac{80}{1\,000\,000} \\ &= \frac{40 + 1280 + 80 + 2 + 200 + 200 + 80}{1\,000\,000 \text{ 小时}} = \frac{1882}{1\,000\,000 \text{ 小时}}\end{aligned}$$

系统 MTTF 为故障率的倒数：

$$\text{MTTF} = \frac{1}{\text{故障率}} = \frac{1\,000\,000 \text{ 小时}}{1882} = 531 \text{ 小时}$$

即基于以上关于 MTTF 的假设，集群平均每三周会发生一次故障。大约 70% 的故障是由磁盘引起的，20% 的故障是由风扇和电源引起的。

6.8 综合：NetApp FAS6000 文件管理器

Network Appliance 公司 1992 年开始进入存储市场，其目标是使用 RAID 4 磁盘阵列和其自身的日志-结构文件系统，提供一种简单易操作的运行 NFS 的文件服务器。该公司之后又加入了对 Window CIFS 文件系统的支持，以及称为行-对角奇偶校验的 RAID 6 策略。为支持应用程序在没有文件系统开销的情况下访问原始数据，例如数据库系统，NetApp 文件管理器能通过标准的光纤通道接口发送数据块。NetApp 也支持 iSCSI，它允许 SCSI 命令在 TCP/IP 网络上运行，这样就可以使用标准的网络设备来连接服务器和存储，例如以太网，从而延长了连接距离。

最新的硬件产品是 FAS6000。它是基于多个 AMD Opteron 微处理器高速通道互连的多处理器。CPU 运行 NetApp 软件栈，包括 NFS，CIFS，RAID-DP 和 SCSI 等。FAS6000 的双 CPU 版是 FAS6030，四 CPU 版是 FAS6070。正如第 4 章提到的，DRAM 分布于 Opteron 的每个微处理器中。FAS6000 的每个 Opteron 处理器连接 8 GB 的 DDR2700，即对于 FAS6030 是 16 GB，对于 FAS6070 是 32 GB。正如第 5 章提到的，DRAM 总线宽度是 128 位，加上用于 SEC/DED 存储的附加位。两种模型都提供了对 I/O 的高速通道。

作为文件管理器，FAS6000 需要大量 I/O 以连到磁盘和服务器。集成的 I/O 由以下组成：

- 8 个光纤通道 (FC) 控制器和端口。
- 8 Gbit 的以太网链路。
- 6 个 8 倍速 (2 GB/s) 的 PCI-E 插槽。

- 3个133 MHz、64位的PCI-X插槽。
- 加上标准的I/O选项，如IDE，USB和32位的PCI。

可以把8个光纤通道控制器放到6个架子上，共包含14个3.5英寸的光纤FC硬盘。这样，集成的I/O中含有磁盘驱动器的最大数量是 $8 \times 6 \times 14 = 672$ 。此外FC控制器还能被放在可选插槽中，以连接最大数目为1008个的驱动器，这样可以减少每个FC网络中驱动器个数并避免竞争的出现等。

也能通过光纤通道SATA磁盘连接到SATA桥控制器，以便允许SATA和FC之间通信。

6个1GB的以太网链路将各个服务器互连起来，如果运行NTFS或CIFS，则FAS6000像一个文件服务器，如果运行iSCSI，则FAS6000又会像一个块服务器。

为了更高的可靠性，FAS6000文件管理器可以成对配置，当一个发生故障时，另一个可以马上接管。集群故障转移要求两个文件管理器通过FC互连成对地访问所有磁盘。这样的互连也允许每个文件管理器在NVRAM中有其他文件管理器的日志数据，同时保持时钟同步。文件管理器的运行状况被实时监控，故障转移过程自动发生。未发生故障的文件管理器保持其自身网络认证和主要功能，但它也同时承担发生故障的文件管理器的网络认证，并通过虚拟文件管理器处理发生故障的文件管理器的所有数据请求，直到管理员把数据服务恢复到初始状态为止。

6.9 谬误和易犯的错误

谬误：组件易出故障。

大部分关于容错的理论都基于这样一种简单的假设：组件在隐含的错误发生之前都可以正常运转；随着故障的发生，组件继而停止运行。

Tertiary Disk项目的情形恰恰与此相反。许多组件在出故障之前的很长一段时间就开始有异常情况出现，而且通常是由系统操作员决定一个组件是否出了故障。直到操作员停止该组件的工作，该组件的运行才会真正终止。

图6.20所示为4种被终止驱动器的记录及其从开始不正常运作到被替换的时间。

系统日志中的故障磁盘信息	日志信息数	故障持续时间（小时）
硬件故障（外围设备写故障）	1763	186
未准备好（诊断故障：ASCQ=组件ID）	1460	90
已恢复差错（超过故障预测阈值）	1313	5
已恢复差错（超过故障预测阈值）	431	17

图6.20 在18个月里Tertiary Disk项目中368个盘中被替换的4个盘的系统日志记录。参见Talagala和Patterson[1999]。这些符合SCSI规范的信息被设备驱动器记录在系统日志中。从信息开始出现到驱动器被替换大约需要一周的时间。第三和第四条信息说明故障即将发生，但其驱动器经过了几个小时才被操作员替换

谬误：如宣传所言，计算机系统已达到99.999%的有效性（“五个九”）。

服务器制造公司的市场销售部门夸大了其计算机硬件的有效性。根据图6.21，可用性为99.999%，简称“五个九”，甚至操作系统公司的市场销售部门也在这样宣传。

每年5分钟不可用可以说是让人满意的结果，但根据市场调查结果显示，这完全不可信。例如，Hewlett-Packard宣称HP-9000服务器硬件和HP-UX操作系统“在预定义、预测试的客户环境下”有

99.999% 的有效性保证 (见[Hewlett-Packard 1998])。这种保证不包括操作者失误、应用程序错误、环境错误等现如今占绝大部分的错误类型。它也不包括预定的停机时间。当然, 如果系统不能得到这种高可靠性保证, 厂商也不会负任何经济责任。

不可用时间 (分钟数/年)	有效性 (%)	有效性级别 (“9” 的个数)
50 000	90	1
5000	99	2
500	99.9	3
50	99.99	4
5	99.999	5
0.5	99.9999	6
0.05	99.999 999	7

图 6.21 实现可用性级别每年不可用时间的分钟数 (来源于 Gary 和 Siewiorek[1991])。注意, “五个九” 意味着每年不可用时间为 5 分钟

微软也声称要争取做到“五个九”。2001 年 1 月, www.microsoft.com 网站的不可用时间为 22 小时。它想要达到 99.999% 的有效性, 需要 250 年的正常使用时间作为不可靠时间的除数。

与市场宣传相反, 2006 年管理良好的服务器的可用性才能达到 99% 或 99.9%。

易犯的错误: 在哪里实现功能会影响到可靠性。

理论上来说, 把 RAID 功能移入软件是件好事。但在实际过程中, 很难保证它运行的可靠性。

软件文化建立在一系列产品发布和连续不断的补丁程序上。分离软件各层是很困难的。例如, 恰当的软件行为基于正常的操作系统版本和补丁程序的发布。因此, 许多用户丢失数据的原因就是软件 bug 或与软件 RAID 系统环境不兼容。

很显然, 虽然硬件系统也会受 bug 的影响, 但硬件系统更强调初始发布的纠错。除此之外, 硬件可靠性基本上与操作系统的版本无关。

谬误: 操作系统是调度磁盘访问的最好地方。

诸如 ATA 和 SCSI 这样高级别的接口向主机操作系统提供了逻辑块地址。在这样的高级别抽象的情况下, 操作系统所能做的就是将逻辑块地址按增序排序。因为只有磁盘知道逻辑块地址到物理扇区、磁道和磁盘面的映射, 这样能减少旋转和寻道时延。

例如, 假设工作负载为四次读操作 [Anderson 2003]:

操作	起始 LBA	长度
读	724	8
读	100	16
读	9987	1
读	26	128

主机会按照逻辑块顺序记录四次读操作:

读	26	128
读	100	16
读	724	8
读	9987	1

按照数据在磁盘上的位置关系记录,对记录顺序的更改可能会更糟糕,如图 6.22 所示。按磁盘调度,读操作需要四分之三圈的磁盘旋转才能完成;按操作系统调度,需要旋转三圈。

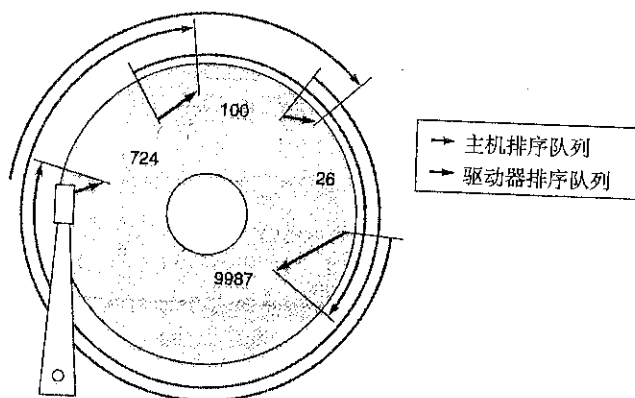


图 6.22 操作系统调度和磁盘调度的访问比较,即主机排序和驱动器排序的对比。前者需要磁盘转 4 圈来完成 3 次读操作,后者只需转 3/4 圈。参考 Anderson[2003]

易犯的错误: 计算机系统的磁盘寻道时间就是寻找 1/3 柱面所用的时间。

这种错误源于不了解厂家是用理想性能来标识磁盘的,并且错误地假设寻道时间是与距离成正比的。以经验规律来看,1/3 距离是从磁盘一个随机位置到另一个随机位置的寻道距离,不包括当前的柱面,且假设磁盘有大量的柱面。过去磁盘厂家标识这个距离是为了与其他磁盘进行比较(现在我们计算“平均值”是用总寻道时间除以数量)。假设(这种假设是不正确的)寻道时间与距离成正比,使用厂商给出的最小和平均寻道时间,可以得出预测寻道时间的一般公式:

$$\text{Time}_{\text{寻道}} = \text{Time}_{\text{最小}} + \frac{\text{距离}}{\text{距离}_{\text{平均}}} (\text{Time}_{\text{平均}} - \text{Time}_{\text{最小}})$$

产生这种错误的原因有两个。首先,寻道时间不是与距离成正比的;磁头臂必须通过加速来克服惯性,先达到其最大速度,在到达指定地点后再减速,直到其停止震动(存取操作时间)。另外,有时磁头臂还必须通过停顿来控制震动。对于超过 200 个柱面的磁盘,Chen 和 Lee[1995]给出了描述寻道距离的模型:

$$\text{寻道时间(距离)} = a \times \sqrt{\text{距离} - 1} + b \times (\text{距离} - 1) + c$$

这里, a , b 和 c 是根据特定磁盘而待定的参数,可以用寻道距离在等于 1、等于最大值和等于三分之一最大值时的寻道时间来求解以上参数。图 6.23 给出了这个公式与错误公式的对照。与第一个等式不同,寻道距离的平方根反映了加速和减速对寻道时间的影响。

第二个问题是,产品说明中平均寻道时间只有当磁盘没有局部性行为时才是准确的。幸运的是,数据访问是有时间局部性和空间局部性的(见附录 C)。例如,图 6.24 给出了两种不同负载条件下的寻道距离情况,条件分别为 UNIX 分时共享的任务负载和商业事务处理负载。可以看到在同一柱面访问(图中用距离 0 表示)的比例较高。因此,这种谬误的误导性强。

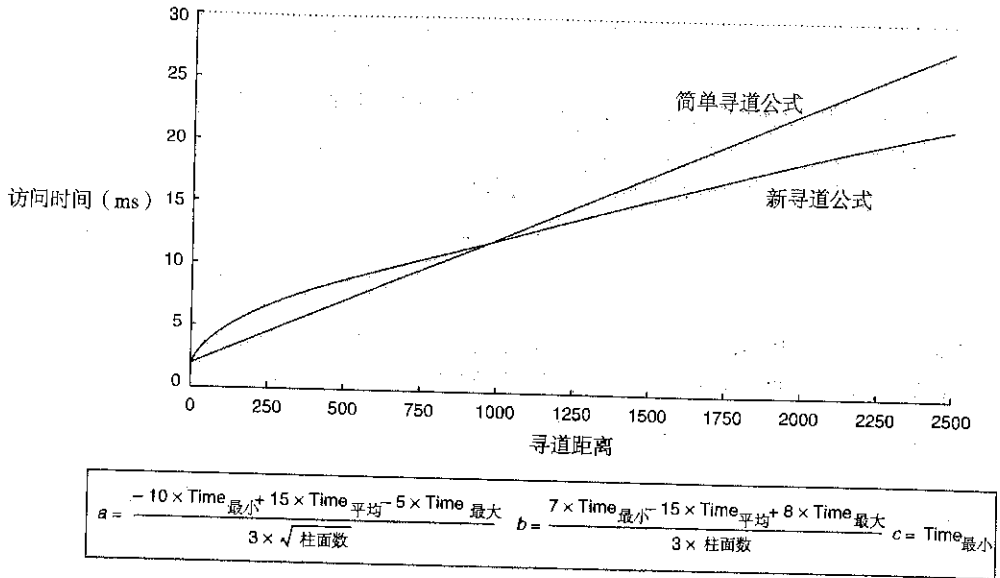


图 6.23 磁盘使用复杂和简单模型的寻道时间和寻道距离之间的关系。Chen和Lee[1995]发现对于一些磁盘来说,上述公式对某些磁盘产品的参数 a , b , c 选择是有很有效的

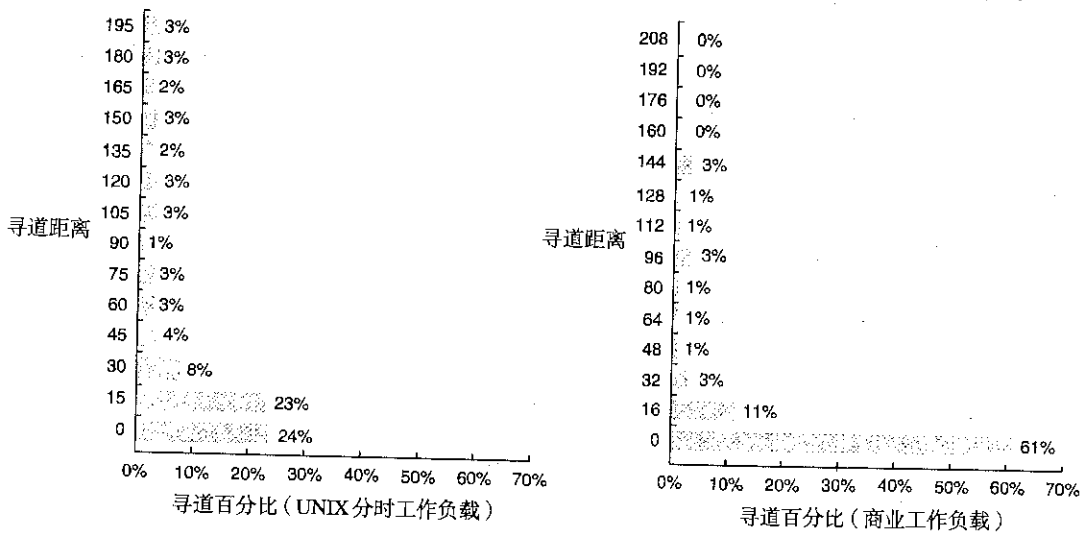


图 6.24 两个系统的寻道距离实例。左图是基于UNIX的分时共享系统。右图基于商业事务处理应用,其磁盘寻道行为是经过调度的。寻道距离为0表示访问在同一柱面进行。其余的数表示y轴坐标上寻道距离所占的百分比。比如右图中的11%横条标注为16,这表示寻道距离在1到16的比例为11%。基于UNIX的图(左图)只表示出1000个柱面中的200个柱面,但其访问所占比例为85%。商业事务处理系统的测试共跟踪了816个柱面。在右图中未能画出来的占访问总数1%或以上的柱面数有224个,所占百分比为4%;304,336,512和624个柱面数所对应的百分比都为1%。图中所示柱面占访问总数的94%,包括其他种类中不同的小数量且非零的访问数。测量数据由Seagate公司的Dave Anderson提供

6.10 结论

存储是一种不易被人们关注的技术。但是，如果看一看现状，我们就会发现存储是真正的王者。有人甚至认为已经变成商品的服务器已成为了存储设备的外围设备。众所周知，根据IBM的估计，在未来2年存储的销售额将超过服务器。

Michael Vizard
Infoworld 主编，2001.8.11]

当存储系统的价值开始凸显时，它们就成为了技术创新和投资的目标。

当今存储系统面临的最大问题是稳定性和可维护性。不仅用户要求数据永不丢失（可靠性），而且当今的应用增加了用户对数据有效访问的要求（有效性）。除了在软硬件可靠性和容错能力上的改进，不易维护对成本和有效性来说都是不容忽视的问题。一个常用的统计数据表明，假设用户花费1美元来购买存储系统，则需要花费6美元到8美元来使用和维护它。当在高级别系统上的多冗余备份无法满足可靠性要求时——例如搜索——则大规模系统会对存储组件的性价比产生影响。

当今 I/O 领域最引人关注的问题是存储系统的稳定性和可维护性。

6.11 历史回顾和参考文献

在随书光盘中的 K.7 节包含了存储设备和技术的发展，其中包括磁盘的发明人、有关 RAID 的发展和操作系统及数据库的历史。此外还附上了进一步的参考文献。

6.12 范例分析及习题^①

范例分析 1：解析磁盘

通过这个示例阐明以下概念：

- 性能特征。
- 微基准测试。

存储系统的内部结构通过一个简单的接口被抽象为一个线性的块阵列。所有存储系统使用一个通用的接口有很多好处：操作系统可以不加区分地使用任何存储器；在不改变接口的情况下，存储系统的内部结构可以任意改变。例如，单个磁盘不管以何种方式实现其最佳性能，都将内部的〈扇区，磁道，表面〉映射到一个线性阵列上；与其类似，多磁盘的 RAID 系统也能将任何数量的磁盘块映射到同样的线性阵列上。但是，固定的接口也有不少缺点。特别是，操作系统在不知道底层存储系统内部分块的准确布局时，就不能实现某些性能、可靠性以及安全等方面的优化。

在此专题中，我们将研究如何通过软件来揭示隐藏在基于块接口下面的存储系统内在结构。基本设想是探查存储系统的特性：通过在存储系统顶层运行一个定义明确的工作负载，并测量不同请求所用的时间，来推断出大量底层系统的细节。

位于 U. C. Berkeley 的 Nisha Talagala 及其同事提出的 Skippy 算法揭示了单个磁盘的一些参数。其中的关键是通过线性增加地址（从 1，2，3 开始增加，依此类推）对单一扇面的

^① 本范例分析由 Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau 提供。

连续访问得到影响磁盘旋转的因素。因此,基本的算法是在每次写操作之前跳过一段磁盘距离,并为每次操作跳过的磁盘距离增加一个扇区的长度,再输出距离和每次写操作的时间。使用原始设备接口的目的是为了避免文件系统优化。SECTOR SIZE 为每次能从磁盘上读出的最小数据量(如 512 字节)。Skippy 算法的详细情况参见 Talagala 等[1999]。

```
fd = open("raw disk device");
for (i = 0; i < measurements; i++) {
    begin_time = gettimeofday();
    lseek(fd, i*SECTOR_SIZE, SEEK_CUR);
    write(fd, buffer, SECTOR_SIZE);
    interval_time = gettimeofday() - begin_time;
    printf("Stride: %d Time: %d\n", i, interval_time);
}
close(fd);
```

通过绘制每次写操作所需时间和寻道距离的函数曲线,可以推断最小传输时间(没有考虑寻道时延和旋转时延)、磁头切换时间、柱面切换时间、旋转时延和磁盘中的磁头数。典型的曲线图有四条截然不同的曲线,它们有相同的斜率和不同的偏移量。最高和最低的曲线与不同的旋转延迟请求相对应,但不包括柱面和磁头切换的开销;两条线的差别揭示出磁盘的旋转时延。次低的曲线与磁头切换请求相对应(另外还加上了旋转时延)。第三条曲线和柱面切换请求相对应(另外也加上了旋转时延)。

6.1 [10/10/10/10] <6.2>图 6.25 表示了在模拟磁盘(磁盘 Alpha)上运行 Skippy 基准程序的输出值。

- [10] <6.2>最小传输时间是多少?
- [10] <6.2>旋转时延是多少?
- [10] <6.2>磁头切换时间是多少?
- [10] <6.2>柱面切换时间是多少?
- [10] <6.2>磁头数是多少?

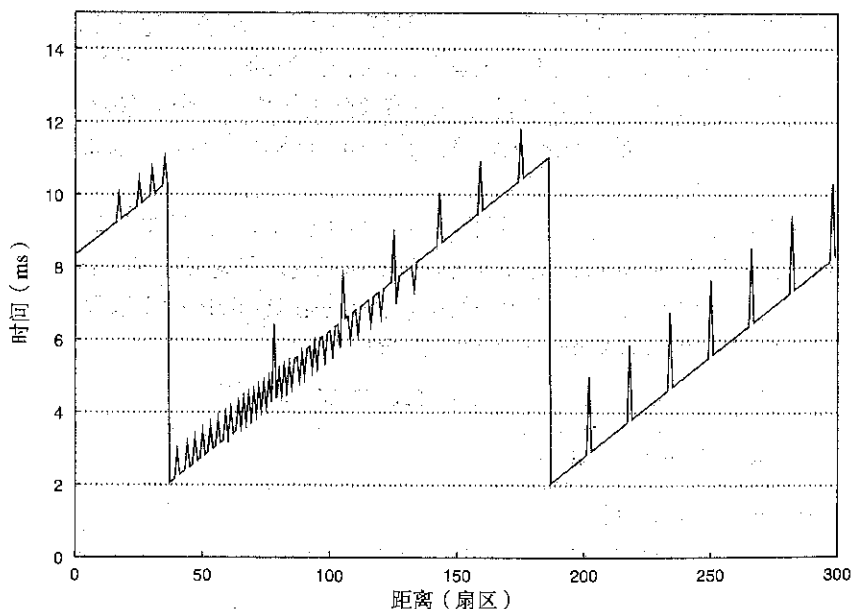


图 6.25 Alpha 磁盘上 Skippy 输出的结果

- 6.2 [25] <6.2>画出磁盘 Beta 上 Skippy 输出的近似曲线图, 磁盘参数如下:
- 最小传输时间: 2.0 ms。
 - 旋转时延: 6.0 ms。
 - 磁头切换时间: 1.0 ms。
 - 柱面切换时间: 1.5 ms。
 - 磁头数目: 4。
 - 每个磁头的扇区数: 100。
- 6.3 [10/10/10/10/10/10] <6.2>选一个磁盘, 并在该磁盘上实现并运行 Skippy 算法。
- a. [10] <6.2>画出 Skippy 输出的曲线图。报告磁盘的模型和生产商。
 - b. [10] <6.2>最小传输时间是多少?
 - c. [10] <6.2>旋转时延是多少?
 - d. [10] <6.2>磁头切换时间是多少?
 - e. [10] <6.2>柱面切换时间是多少?
 - f. [10] <6.2>磁头数是多少?
 - g. [10] <6.2>Skippy 程序运行在实际的磁盘上和运行在假想的磁盘上在结果上有什么定性的区别?

范例分析 2: 解析磁盘阵列

通过这个示例阐明以下概念:

- 性能特征。
- 微基准测试。

Wisconsin 大学的 Timothy Denehy 和其同事提出的 Shear 算法[Denehy 等 2004]给出了 RAID 系统的参数。其基本思想是向 RAID 阵列产生请求作为工作负载, 然后记录这些请求的时间。通过观察每组请求时间的长短, 能够推断出哪些块被分配到同一个磁盘上。

我们将 RAID 的属性定义如下。RAID 中的数据是以块为单位分配给磁盘的, 这里的块是文件系统从存储系统中读写的最小数据单元。因此, 块的大小由文件系统和指纹识别软件决定。一个大块 (chunk) 是在磁盘上一组连续分配的块。一个数据带 (stripe) 是数据磁盘上的一组大块。最小的连续数据块称为 pattern, 这样在 pattern 中块的偏移量 i 总是位于磁盘 j 上。

- 6.4 [20/20] <6.2>通过以下的代码能揭示出 pattern 的大小。以下代码在没有文件系统优化的情况下访问原始设备。Shear 算法的关键是使用随机的请求来避免引起任何预取以及在 RAID 中或单个磁盘中的缓存机制。此代码的基本思想是在 RAID 阵列中以固定的间隔 p 连续访问 N 个随机块, 并测量每个间隔所用的时间。

```
for (p = BLOCKSIZE; p <= testsize; p += BLOCKSIZE) {
    for (i = 0; i < N; i++) {
        request[i] = random()*p;
    }
    begin_time = gettimeofday();
    issues all request[N] to raw device in parallel;
    wait for all request[N] to complete;
```

```

interval_time = gettimeofday() - begin_time;
printf("PatternSize: %d Time: %d\n", p,
      interval_time);
}

```

如果将此代码在 RAID 阵列上运行, 将 N 次请求的测量时间作为 p 的函数并画图, 则当所有 N 次请求指向同一块磁盘时, 时间的数值最高。因此, 最高的时间值 p 对应于 RAID 中的 pattern 大小。

a. [20] <6.2>图 6.26 指出了在一个未知的 RAID 系统中运行 pattern 大小算法的结果。

- 此存储系统的 pattern 的大小是多少?
- 在此存储系统中, 0.4, 0.8 和 1.6 的测量时间对应的是什么?
- 如果是 RAID 0 阵列, 则系统中有多少个磁盘?
- 如果是 RAID 0 阵列, 则 chunk 的大小是多少?

b. [20] <6.2>画出在以下磁盘系统上运行 Shear 代码的结果, 磁盘参数如下:

- 请求数: $N = 1000$ 。
- 随机读磁盘的时间: 5 ms。
- RAID 级别: RAID 0。
- 磁盘数量: 4。
- chunk 大小: 8 KB。

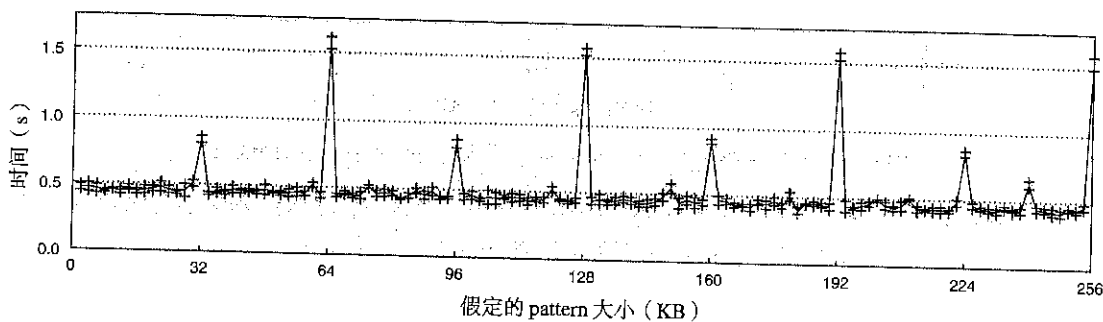


图 6.26 在假定磁盘上运行 Shear 算法计算 pattern 大小输出的结果图

6.5 [20/20] <6.2>通过以下的代码能揭示出 chunk 的大小。基本思想是在选定的随机 N 个 pattern 上进行读操作, 但在 pattern 内控制偏移量 c 和 $c-1$ 。

```

for (c = 0; c < patternsizes; c += BLOCKSIZE) {
    for (i = 0; i < N; i++) {
        requestA[i] = random()*patternsizes + c;
        requestB[i] = random()*patternsizes +
            (c-1)*patternsizes;
    }
    begin_time = gettimeofday();
    issue all requestA[N] and requestB[N] to raw device
        in parallel;
    wait for requestA[N] and requestB[N] to complete;
    interval_time = gettimeofday() - begin_time;
    printf("ChunkSize: %d Time: %d\n", c, interval_time);
}

```

如果运行此段代码, 将测量时间作为 c 的函数并画图, 则当 requestA 和 requestB 读操作指向不同的两块磁盘时, 测量时间最低。因此, 最低的 c 值对应于 RAID 的磁盘间 chunk 的边界值。

a. [20] <6.2>图 6.27 指出在一个未知的 RAID 系统中, 运行计算 chunk 大小的算法的结果。

- 此存储系统的 chunk 大小是多少?
- 在此存储系统中, 0.75 和 1.5 的测量时间对应什么?

b. [20] <6.2>画出在以下磁盘系统上运行 Shear 代码的结果图, 磁盘参数如下:

- 请求数: $N = 1000$ 。
- 随机读磁盘的时间: 5 ms。
- RAID 级别: RAID 0。
- 磁盘数量: 8。
- chunk 大小: 12 KB。

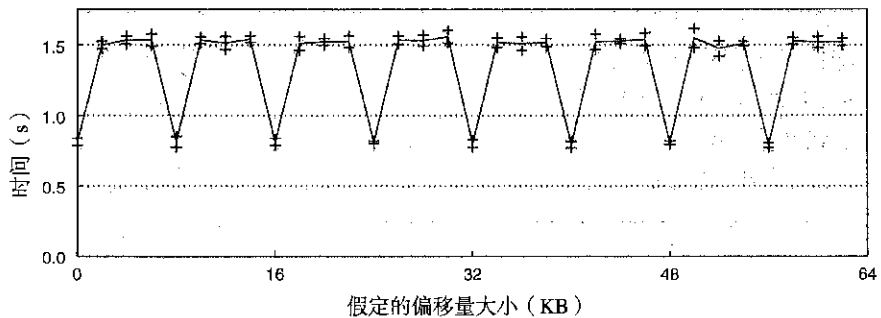


图 6.27 在假定磁盘上运行 Shear 算法计算 chunk 大小输出的结果图

6.6 [10/10/10/10] <6.2>通过以下的代码能揭示出磁盘上 chunk 的布局。基本思想是选择 N 个随机 pattern, 并彻底读出 pattern 上两两合并的 chunk。

```
for (a = 0; a < numchunks; a += chunksize) {
    for (b = a; b < numchunks; b += chunksize) {
        for (i = 0; i < N; i++) {
            requestA[i] = random()*patternsize + a;
            requestB[i] = random()*patternsize + b;
        }

        begin_time = gettimeofday();
        issue all requestA[N] and requestB[N] to raw device
        in parallel;
        wait for all requestA[N] and requestB[N] to
            complete;

        interval_time = gettimeofday() - begin_time;
        printf("A: %d B: %d Time: %d\n", a, b,
            interval_time);
    }
}
```

在运行这段代码后, 可以得到测量时间作为 a 和 b 的函数。最简单的画图方法是以 a 和 b 为参数建立一个二维表, 用阴影的值来表示时间。使用更深的阴影表示更快的时间, 更浅的阴影表示更慢的时间。因此, 浅色的阴影表明 pattern 中的偏移量 a 和 b 指向了同一个磁盘。

图 6.28 显示了在 pattern 大小为 384 KB、chunk 大小为 32 KB 的存储系统上运行划分算法的结果。

- [20] <6.2> 在 pattern 中有多少个 chunk?
- [20] <6.2> 每个 pattern 中的哪些 chunk 是分配在同一磁盘上的?
- [20] <6.2> 此存储系统中有多少个磁盘?
- [20] <6.2> 画出磁盘上可能的块划分?

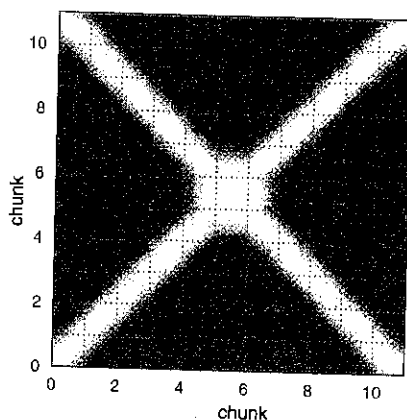


图 6.28 在假定磁盘上运行 Shear 算法计算 chunk 划分的输出结果图

- 6.7 [20] <6.2> 画出在如图 6.29 所示的磁盘系统上, 运行划分算法的结果图。此存储系统有 4 块磁盘, chunk 的大小为 4 个 4 KB 的块 (即 16 KB), 使用 RAID 5 左 - 非对称分布。

00	01	02	03	04	05	06	07	08	09	10	11	P	P	P	P
12	13	14	15	16	17	18	19	P	P	P	P	20	21	22	23
24	25	26	27	P	P	P	P	28	29	30	31	32	33	34	35
P	P	P	P	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	P	P	P	P
60	61	62	63	64	65	66	67	P	P	P	P	68	69	70	71
72	73	74	75	P	P	P	P	76	77	78	79	80	81	82	83
P	P	P	P	84	85	86	87	88	89	90	91	92	93	94	95

Parity: RAID 5 Left-Asymmetric, stripe = 16, pattern = 48

图 6.29 有 4 块磁盘、chunk 的大小为 4 个 4 KB 的块 (即 16 KB)、使用 RAID 5 左 - 非对称分布的磁盘系统。图中所示为 pattern 的两个副本

范例分析 3: RAID 重构

通过这个示例阐明以下概念:

- RAID 系统。
- RAID 的重构。
- 平均故障时间 (MTTF)。
- 直到数据丢失的平均时间 (MTDL)。
- 可执行性。
- 双故障。

当磁盘发生故障时, RAID 系统要确保数据不会丢失。因此 RAID 最重要的功能是在磁盘故障时, 对数据重建。这个过程称为**重构**, 即本专题讨论的主要内容。读者将需要同时考虑 RAID 系统能处理 1 个磁盘发生故障的情况, 以及 RAID-DP 系统能处理 2 个磁盘发生故障的情况。

重构通常有两种不同的实施方式。在**离线重构**中, RAID 把所有资源用来执行重构, 且不接受任何工作负载的服务请求。在**在线重构**中, RAID 在执行重构的同时, 继续接受工作负载的服务请求; 重构的过程只能使用有限的 RAID 系统总带宽。

但实施重构会影响系统的**可靠性**和**可执行性**。在 RAID 5 中, 如果在第一个磁盘的数据恢复之前第二个磁盘发生故障, 则会导致数据丢失。即重构时间 (MTTR) 越长, 可靠性或直到数据丢失的平均时间 (MTDL) 越低。可执行性是同时考虑系统性能和可靠性时的指标; 它被定义为一个给定状态的系统性能乘以此状态出现的概率。对于 RAID 磁盘阵列来说, 可能的状态包括无磁盘故障的正常操作状态、单磁盘故障的重构状态和多磁盘故障的关闭状态。

对于以下习题, 假设由 6 块磁盘构成 RAID 系统, 另外还有足够的剩余空间。假设每个磁盘为 37 GB 的 SCSI 磁盘, 如图 6.3 所示。假设每个磁盘能以 142 MB/s 的峰值速率连续读数据, 并以 85 MB/s 的峰值速率连续写数据。磁盘通过 Ultra320 的 SCSI 总线互连, 能以总共 320 MB/s 的速率传输。假设每个磁盘的故障是独立的, 并忽略系统中的其他故障。对于重构过程, 可以假设任何异或 (XOR) 计算和存储器复制的开销可以忽略。在在线重构期间, 假设重构过程受 RAID 系统的总带宽 10 MB/s 的限制。

- 6.8 [10] <6.2> 假设 6 个磁盘组成 RAID 4 系统。画出简单的图表以表示此系统中磁盘上块的划分。
- 6.9 [10] <6.2, 6.4> 当单磁盘故障时, RAID 4 系统会进行重构。重构所需的预期时间是多少?
- 6.10 [10/10/10] <6.2, 6.4> 假设 RAID 4 重构的起始时间为 t 。
- a. [10] <6.2, 6.4> 进行重构时需要执行什么读写操作?
 - b. [10] <6.2, 6.4> 对于离线重构, 何时会全部完成?
 - c. [10] <6.2, 6.4> 对于在线重构, 何时会全部完成?
- 6.11 [10/10/10/10] <6.2, 6.4> 此题中, 我们将研究直到数据丢失的平均时间 (MTDL)。在 RAID 4 中, 仅在第一个磁盘故障修复前发生第二次磁盘故障时, 才会发生数据丢失。
- a. [10] <6.2, 6.4> 在离线重构期间, 发生第二次磁盘故障的可能性有多大?
 - b. [10] <6.2, 6.4> 考虑在重构期间发生第二次磁盘故障的可能性, 则对于离线重构, MTDL 是多少?
 - c. [10] <6.2, 6.4> 在在线重构期间, 发生第二次磁盘故障的可能性有多大?
 - d. [10] <6.2, 6.4> 考虑在重构期间发生第二次磁盘故障的可能性, 则对于在线重构, MTDL 是多少?
- 6.12 [10] <6.2, 6.4> RAID 4 阵列对于离线重构的可执行性是多少? 使用 IOPS 计算可执行性, 假设随机的只读工作负载平均分布于磁盘阵列中。
- 6.13 [10] <6.2, 6.4> RAID 4 阵列对于在线重构的可执行性是多少? 在线修复期间, 假设 IOPS 降到其峰值的 70%。是离线还是在线重构的可执行性较好?
- 6.14 [10] <6.2, 6.4> RAID 6 用来解决两个并发的磁盘故障。假设已有基于行-对角奇偶校验的 RAID 6 系统, 即 RAID-DP; 而且 6 个磁盘的 RAID-DP 系统是基于 $p=5$ 的 RAID 4, 如

图6.5所示。如果磁盘0和磁盘3发生故障,这些磁盘如何重构?指出在前面4个数据带上计算缺失块的所需步骤。

范例分析 4: RAID 性能的预测

通过这个示例阐明以下概念:

- RAID 级别。
- 排队论。
- 工作负载的影响。
- 磁盘划分的影响。

在此专题中,我们将研究如何将简单排队论用于预测 I/O 系统的性能。除此之外,还会研究存储系统配置和工作负载是如何影响服务时间、磁盘利用率和平均响应时间的。

存储系统的配置对性能有较大影响。不同的 RAID 级别能用不同的方法通过排队论建模。例如,在假设请求是在 N 个磁盘上适当分布的前提下,包含 N 个磁盘的 RAID 0 系统能用 N 个独立的 M/M/1 队列系统建模。RAID 1 阵列的行为依赖于工作负载:读操作可以交给镜像磁盘来做,但写操作必须同时交给两个盘。因此,对于只读的工作负载,2 个磁盘的 RAID 1 阵列能用 M/M/2 队列建模;对于只写的工作负载,要用 M/M/1 队列建模。包含 N 个磁盘的 RAID 4 阵列的行为也取决于工作负载:读操作可以只交给一个特殊的数据盘,但写操作必须更新所有的奇偶校验盘,这也是系统性能的瓶颈。因此,对于只读的工作负载,RAID 4 可以视为 $N-1$ 个单独的系统建模,对于只写的工作负载,用 M/M/1 队列建模。存储系统中块的划分对性能也有重大影响。考虑一个 40 GB 容量的单一磁盘,如果工作负载随机访问 40 GB 的数据,则块的划分对性能不会产生多大影响。但如果只随机访问一半容量的数据(20 GB 的数据),则块的划分确实会对结果产生影响:要减小寻道时间,可以将 20 GB 的数据紧凑地放在 20 GB 连续的磁道上,而不是统一分配在整个 40 GB 容量上。

对于此问题,我们将使用简化的模型来评估磁盘的服务时间。在这个基本模型中,对于小的随机访问请求的平均配置和传输时间是寻道距离的线性函数。对 40 GB 的磁盘,假设服务时间为 $5 \text{ ms} \times \text{空间利用率}$ 。则如果整个 40 GB 的磁盘被用完,随机访问请求的平均配置和传输时间就是 5 ms;如果只使用了 20 GB,则随机访问请求的平均配置和传输时间就是 2.5 ms。

在此范例分析中,可以假设处理器每分钟发送 167 个小的随机磁盘访问请求且这些请求都呈指数分布。假设请求的大小等于块大小,为 8 KB。系统中每个磁盘都是 40 GB。不考虑存储系统的配置,工作负载访问总共 40 GB 的数据;可以使用最有效的方法分配系统中 40 GB 的磁盘数据。

6.15 [10/10/10/10/10] <6.5> 首先假设此存储系统由一块 40 GB 的磁盘组成。

- [10] <6.5> 给定的工作负载和存储系统,其服务时间是多少?
- [10] <6.5> 磁盘的平均利用率是多少?
- [10] <6.5> 每次请求的平均磁盘等待时间为多少?
- [10] <6.5> 队列中的平均请求次数是多少?
- [10] <6.5> 最后,磁盘请求的平均响应时间是多少?

- 6.16 [10/10/10/10/10/10] <6.2, 6.5> 设想存储系统包含 2 块 40 GB 的磁盘, 以 RAID 0 组成阵列, 即数据以 8 KB 的块无冗余地带状分布在两块磁盘上。
- [10] <6.2, 6.5> 40 GB 的数据如何分配在磁盘上? 在总共 40 GB 的容量上给定一个随机请求, 每次请求的预计时间是多少?
 - [10] <6.2, 6.5> 如何将排队论用于此存储系统的建模?
 - [10] <6.2, 6.5> 每个磁盘的平均利用率是多少?
 - [10] <6.2, 6.5> 每次请求的平均磁盘等待时间为多少?
 - [10] <6.2, 6.5> 队列中的平均请求次数是多少?
 - [10] <6.2, 6.5> 最后, 磁盘请求的平均响应时间是多少?
- 6.17 [20/20/20/20/20] <6.2, 6.5> 设想存储系统包含 2 块 40 GB 的磁盘, 以 RAID 1 组成阵列, 即数据在两块磁盘上形成镜像。为只读的工作负载对系统使用排队论建模。
- [20] <6.2, 6.5> 40 GB 的数据如何分配在磁盘上? 在总共 40 GB 的容量上给定一个随机请求, 每次请求的预计时间是多少?
 - [20] <6.2, 6.5> 如何将排队论用于此存储系统的建模?
 - [20] <6.2, 6.5> 每个磁盘的平均利用率是多少?
 - [20] <6.2, 6.5> 每次请求的平均磁盘等待时间为多少?
 - [20] <6.2, 6.5> 最后, 磁盘请求的平均响应时间是多少?
- 6.18 [10/10] <6.2, 6.5> 如上题假设, 为只写的工作负载对系统使用排队论基于 RAID 1 建模。
- [10] <6.2, 6.5> 描述如何使用排队论为此系统和工作负载建模?
 - [10] <6.2, 6.5> 对于给定的系统和工作负载, 平均利用率、平均等待时间和平均响应时间是多少?

范例分析 5: I/O 子系统设计

通过这个示例阐明以下概念:

- RAID 系统。
- 平均故障时间 (MTTF)。
- 性能和可靠性之间的折中。

在此专题中, 我们要在给定的预算下设计 I/O 子系统。系统的容量不能低于最小要求的值, 而且需要对性能和可靠性进行优化。可在预算之内自由选择多种磁盘和控制器。以下是可选的组成模块:

- 10 000 MIPS 的 CPU 花费: \$1000。其 MTTF 为 1 000 000 小时。
- 1000 MB/s 的 I/O 总线, 包括 20 个 Ultra320 SCSI 总线和控制器。
- Ultra320 SCSI 总线的传输率为 320 MB/s, 每根总线能支持 15 个磁盘 (称为 SCSI 串)。SCSI 电缆的 MTTF 为 1 000 000 小时。
- Ultra320 SCSI 控制器性能: 50 000 IOPS 花费 \$250, MTTF 为 500 000 小时。
- 机箱电源和 8 个磁盘的冷却设备需 \$2000。机箱的 MTTF 为 1 000 000 小时, 风扇的 MTTF 为 200 000 小时, 电源的 MTTF 为 200 000 小时。
- SCSI 磁盘如图 6.3 所示。
- 更换任何故障元件需要 24 小时。

可以对工作负载做以下假设：

- 对每个磁盘 I/O，操作系统需要 70 000 个 CPU 指令周期。
- 工作负载由许多并发的随机 I/O 组成，平均大小为 16 KB。

你所构建的系统必须满足以下特性：

- 预算不超过 28 000 美元。
- 容量至少为 1 TB。

- 6.19 [10] <6.2>首先设计一个 I/O 子系统，仅优化容量和性能（不考虑可靠性），有明确的 IOPS。讨论哪种 RAID 的级别和块大小能带来最佳的性能。
- 6.20 [20/20/20/20] <6.2, 6.4, 6.7>在给定的预算和容量条件下，哪种 SCSI 磁盘、控制器和机箱能实现最佳性能？
- a. [20] <6.2, 6.4, 6.7>系统中预期的 IOPS 是多少？
 - b. [20] <6.2, 6.4, 6.7>系统总共花费多少？
 - c. [20] <6.2, 6.4, 6.7>系统容量是多少？
 - d. [20] <6.2, 6.4, 6.7>系统的 MTTF 是多少？
- 6.21 [10] <6.2, 6.4, 6.7>现在重新设计系统来优化可靠性，创建 RAID 10 或 RAID 01 磁盘阵列。此存储系统不仅对磁盘故障具有强大的处理能力，还能应对控制器、电缆、电源和风扇的故障；特别是单独的组件故障不会影响对一对副本的访问。画图说明有多少块分配到 RAID 10 或 RAID 01 的配置中。是 RAID 10 还是 RAID 01 磁盘阵列更适合此环境？
- 6.22 [20/20/20/20] <6.2, 6.4, 6.7>要优化 RAID 10 或 RAID 01 磁盘阵列以提高可靠性（仍然保持预算和容量的要求），应该用哪种 RAID 配置？
- a. [20] <6.2, 6.4, 6.7>系统中组件的全局 MTTF 是多少？
 - b. [20] <6.2, 6.4, 6.7>系统的 MTDL 是多少？
 - c. [20] <6.2, 6.4, 6.7>系统可用的容量是多少？
 - d. [20] <6.2, 6.4, 6.7>假设一个只写的工作负载，预期的 IOPS 是多少？
- 6.23 [10] <6.2, 6.4, 6.7>假设现在能够得到一个有两倍容量的磁盘，但价格不变。如果仍只考虑可靠性，将如何改变存储系统的配置？请陈述原因。

范例分析 6：失效位

通过这个示例阐明以下概念：

- 局部磁盘故障。
- 故障分析。
- 性能分析。
- 奇偶校验保护。
- 校验和。

现在要求读者负责处理避免“位失效”的问题——文件中的块或位由于时间而失效。这个问题在归档场景下是特别重要的，当数据一次写入之后，可能很多年后才会被访问；如果不采用保护数据的额外措施，就会因为介质错误或其他 I/O 故障使得文件中的位或块慢慢改变或变得不可用。

处理位失效需要两个特殊的组件：探测和恢复。为有效探测位失效，可以在文件的每个块上使用校验和；校验和是某种类型的函数，输入是一串数据（长度不定），输出是一串固定长度的数据（校验和）。如果数据发生改变，计算得出的校验和也会相应变化。

一旦检测到校验和变化，就需要某种形式的冗余来恢复位失效。例如镜像（保持每个块的多个副本）和奇偶校验（某些额外的冗余信息，通常比镜像有更高的空间有效性）。

在此专题的研究中，我们将分析不同场景下各种技术的有效性。我们还会编写代码以在一组文件上实现数据完整性的保护。

- 6.24 [20/20/20] <6.2>假设在习题 6.24 至习题 6.27 中，将使用简单的奇偶校验保护。假设在文件系统中为每个文件计算一个奇偶校验块。另外，假设每个文件中每 4 KB 的块用 20 字节的 MD5 做校验和。首先，解决空间开销上的问题。依据最近的研究[Douceur 和 Bolosky 1999]，在现代的 PC 中可以发现不同文件大小的分布如下：

≤1 KB	2 KB	4 KB	8 KB	16 KB	32 KB	64 KB	128 KB	256 KB	512 KB	≥1 MB
26.6%	11.0%	11.2%	10.9%	9.5%	8.5%	7.1%	5.1%	3.7%	2.4%	4.0%

研究也发现文件系统通常是半满的。假设有一个 37 GB 的磁盘卷基本装满一半，文件依照同样的分布，请回答以下问题：

- [20] <6.2>为能够检测校验和的单一错误，需要在磁盘上保存多少额外信息（包括字节中的和磁盘卷的百分比）？
 - [20] <6.2>为能够检测并纠正校验和的单一错误，需要在磁盘上保存多少额外信息（包括字节中的和磁盘卷的百分比）？
 - [20] <6.2>文件大小分布如上，使用块大小计算校验和是否太大、太小或者正好？
- 6.25 [10/10] <6.2, 6.3>数据保护中存在的最大问题是错误检测。一种方法是延迟执行错误检测，即直到文件被访问时才检查并确保正确的数据。这种方法的问题是并不被频繁访问的文件可能会慢慢失效，而当需要访问时，又有很多的错误需要纠正。因此，另一种主动的方法又称为磁盘擦洗，即定期检查数据以便提前发现错误。
- [10] <6.2, 6.3>假设位抖动独立发生，频率为每个月 1 GB 的数据有 1 位抖动。假设同样 20 GB 的卷为半满的，使用如图 6.3 所指定的 SCSI 磁盘（4 ms 的寻道时间，100 MB/s 的传输率），需要多长时间来扫描文件以检测和修复其完整性？
 - [10] <6.2, 6.3>什么样的位抖动率会使保持数据完整性变得不可能？仍然假设 20 GB 的卷和 SCSI 磁盘。
- 6.26 [10/10/10/10] <6.2, 6.4>加入数据保护后可能会增加性能上的开销。现在研究这种数据保护方法的性能开销。
- [10] <6.2, 6.4>假设在 SCSI 磁盘上连续写一个 40 MB 的文件，则为实现数据保护策略需写入额外的信息到磁盘上。此策略需要产生多少写流量（包括字节卷的总数和作为总流量的百分比）？
 - [10] <6.2, 6.4>假设现在需要随机更新文件，文件和数据库表相似。即假设执行一连串的随机写文件操作，每个时间执行一个单独的写操作，而且必须更新盘上的保护信息。假设执行了 10 000 次随机写操作，该策略需要产生多少 I/O 流量（包括字节卷的总数和作为总流量的百分比）？

c. [10] <6.2, 6.4>假设数据保护信息总是保存在磁盘的一个独立部分, 远离所保护的文件 (即对每个文件 A, 有另一个文件 $A_{\text{checksums}}$ 为 A 的校验和)。因此, 在读操作时可能会有开销——即每次读时, 需要用校验和来探测数据是否失效。

假设从磁盘上连续地读取 10 000 个 4 KB 大小的块。以 4 ms 的寻道时间和 100 MB/s 的传输率 (如图 6.3 所示), 从磁盘上读取文件 (和相应的校验和) 需要多长时间? 加入校验和后的损失时间是多少?

d. [10] <6.2, 6.4>假设数据保护信息总是保存在磁盘的一个独立部分, 和 c 中类似, 假设从一个很大的文件中连续地读取 10 000 个 4 KB 大小的块 (即该文件远远大于 10 000 个块)。

对每次读, 需要再次使用校验和来确保数据完整性。假设有同样的磁盘特性, 那么从磁盘上读这 10 000 个块需要多长时间? 加入校验和后的损失时间是多少?

6.27 [40] <6.2, 6.3, 6.4>最后, 我们将理论用于实践来开发一款用户级的工具以预防文件失效。假设你需要编写一组简单的工具来探测和修复文件的完整性。第一个工具是用来做校验和以及奇偶校验的。它称为 build, 按如下格式使用:

```
build <filename>
```

build 程序会为 filename 文件生成一个称为 file-name.cp 的文件, 包含所需的校验和及冗余信息, 并保存在同一目录下 (以便之后容易找到)。

第二个程序被用来检查和修复潜在的被损坏的文件。它称为 repair, 按如下格式使用:

```
repair <filename>
```

repair 程序需要参考 .cp 文件的文件名, 并检验所有存储的校验和与数据计算得到的校验和一致。如果校验和对一个块不匹配, repair 会使用冗余信息重建正确的数据并修复文件。如果二个或以上的块损坏, repair 只是简单地报告文件失效。为测试系统, 我们提供一个称之为 corrupt 的工具, 用来使文件失效。按如下格式使用:

```
corrupt <filename> <blocknumber>
```

corrupt 使用随机噪声来将指定的文件块号失效。对于使用 MD 5 的校验和, 它将输入串并生成一个 128 位的“指纹”或校验和作为输出。MD 5 的一个简单实现如下:

http://sourceforge.net/project/showfiles.php?group_id=42360

奇偶校验使用异或操作计算。用 C 程序可以计算两个块的奇偶校验, 每个块的长度为 BLOCKSIZE, 具体如下:

```
unsigned char block1[BLOCKSIZE];
unsigned char block2[BLOCKSIZE];
unsigned char parity[BLOCKSIZE];

// 首先清空奇偶校验块
for (int i = 0; i < BLOCKSIZE; i++)
    parity[i] = 0;

// 然后计算奇偶校验, C 中异或的标示为 ^
for (int i = 0; i < BLOCKSIZE; i++) {
    parity[i] = block1[i] ^ block2[i];
}
```

范例分析 7: 排序

通过这个示例阐明以下概念:

- 基准程序测试。
- 性能分析。
- 性价比分析。
- 分摊开销。
- 系统平衡。

在数据库领域使用基准测试程序来评价系统性能是常用的方法。在此专题中,我们将研究由 Anonymous 等[1985]给出的一种测试方法(见第1章):外部的或磁盘到磁盘的排序。由于多种原因,排序已成为一种令人关注的基准测试方法。首先,排序需要使用到计算机系统的所有组件,包括磁盘、存储器和处理器。其次,在最高性能的排序上需要大量有关 CPU Cache、操作系统和 I/O 子系统的专门技术。最后,它对于学生来说易于实现。依据数据量的大小,排序都能在一个或多个阶段中完成。简单地讲,如果有足够大的存储器能保存整个数据集,就能将整个数据集读入存储器,对其排序,然后写回。这称为“一次性”排序。

如果没有足够大的存储器,数据排序需要历经多个阶段。有许多不同的实现方法。一种简单的方法是对每个输入文件的 chunk 排序并写入磁盘;将已排序的文件保留在磁盘上。然后将每个已排序的临时文件合并,得到最终排序文件。这称为“两次性”排序。多个阶段的排序用在不能于第二个阶段就将所有的数据流合并的情况。

在此专题的研究中,我们将分析排序的各个不同方面,决定其在不同场景中的有效性和成本-有效性。还会编写外部排序,来测量实际硬件上的性能。

- 6.28 [20/20/20] <6.4>作为第一步,我们首先对一个系统进行配置,使其能在最短的时间内完成排序,不必考虑开销上的限制。为了使排序能够顺利完成,应确保系统能提供充足的峰值带宽。

假设执行内存排序的时间与给定的机器的 CPU 时钟频率及存储器带宽成正比(例如,对机器上 1 MB 的记录排序,使用 1 MB/s 的存储器带宽和 1 MIPS 的处理器需要花 1 分钟)。并假设排序的 I/O 状态能获得连续的带宽。显然,如果没有足够的存储器装载所有数据以进行一次性排序,那么就需要两次排序。

在执行 I/O 时你可能会遇到一个问题:系统时常会产生额外的**存储器副本**;例如,当系统调用 read() 时,数据首先会从磁盘读入系统缓存中,随后复制到指定的用户缓存中。因此, I/O 期间的存储器带宽会产生问题。

最后,假设没有读、排序和写的重叠。即当从磁盘读数据时,只做此操作;当进行排序时,完全利用 CPU 和存储器带宽。当做写操作时,只将数据写到磁盘。

你的任务是配置一个系统,为 1 GB 数据(如大约 1000 万个 100 字节的数据)的排序获得峰值性能。参考下表选择需要使用的机器、存储器、I/O 互连和磁盘。

注意,假设购买的是单处理器系统,且最多可以有 2 个 I/O 互连。但存储器和磁盘的数量取决于你的需要(假设每个 I/O 互连没有磁盘数量上的限制)。

- a. [20] <6.4>机器总的费用是多少(分成各个组成部分,包括 CPU 的花费、存储器数量、磁盘数和 I/O 总线)?

- b. [20] <6.4>执行完成1 GB数据的排序需要多长时间(分成读磁盘的时间、写磁盘的时间和排序时间)?
- c. [20] <6.4>系统的性能瓶颈是什么?

CPU			I/O 互连		
慢速	1 GIPS	\$200	慢速	80 MB/s	\$50
标准	2 GIPS	\$1000	标准	160 MB/s	\$100
快速	4 GIPS	\$2000	快速	320 MB/s	\$400
存储器			磁盘		
慢速	512 MB/s	\$100/GB	慢速	30 MB/s	\$70
标准	1 GB/s	\$200/GB	标准	60 MB/s	\$120
快速	2 GB/s	\$500/GB	快速	110 MB/s	\$300

6.29 [25/25/25] <6.4>现在将分析排序的性价比。毕竟买到一款高性能的机器很容易,但找到性价比高的却不容易。

PennySort 竞赛 (research.microsoft.com/barc/SortBenchmark) 中就讨论了这一问题。PennySort问是否能只用1美分排序尽可能多的数据。为回答此问题,需要假设所购买的系统能使用三年(94 608 000秒),用机器总的花费(美分)除以时间,可以得到每分钱的时间预算。

这里的任务更简单。假设有固定的\$2000预算(或更少)。能构建怎样的最快排序的机器?使用习题6.28的表中列出的硬件来配置机器。

- a. [25] <6.4>机器总的费用是多少(分成各个组成部分,包括CPU的花费、存储器数量、磁盘数和I/O总线)?
- b. [25] <6.4>如何分配读磁盘的时间、写磁盘的时间和排序时间?
- c. [25] <6.4>系统的性能瓶颈是什么?
- 6.30 [20/20/20] <6.4, 6.6>要获得较好的磁盘性能通常需要分摊开销。其基本思想是:如果必须承受某种开销,那么尽可能将花费用来做最有用事,从而会减少影响。这个思想普遍用在计算机的多个领域中。对于磁盘,在传输数据前,会有寻道和旋转开销。可以通过传输一个大块的数据来分摊寻道和旋转的开销。
- 在此题中,我们关注如何在第二次的二次排序中分摊寻道和旋转开销。假设当第二次排序开始时,磁盘上有 N 个排序,每个的大小都刚好填满内存。这里的任务是从每个排好序的数据中读一个chunk并将结果合并成一个最终的输出排序。注意每次读都会有寻道和旋转,因为上一次读可能会在不同的数据流中。
- a. [20] <6.4, 6.6>假设有一磁盘,传输率为100 MB/s,平均寻道开销是7 ms,转速为10 000 rpm。并假设每次从一个数据流中读1 MB的数据,每1 GB的容量有100个数据流。假设写(最终的排序输出)发生在1 GB的chunk上。假设I/O是主要的开销,合并需要多长时间?
- b. [20] <6.4, 6.6>假设改变每次读的大小,从1 MB变为10 MB。执行第二次排序的总时间是多少?
- c. [20] <6.4, 6.6>在两种情况下,假设我们希望最大化磁盘有效率。在磁盘访问的全部时间内,以传输数据的有效时间率作为磁盘的有效率。上面提到的每个场景的磁盘有效率各为多少?

- 6.31 [40] <6.2, 6.4, 6.6> 本题需要读者自己编写外部排序程序。为产生数据组，我们提供一个称为 generate 的工具，按如下格式使用：

```
generate <filename> <size (in MB)>
```

运行 generate 会创建一个名为 filename、大小为 size 的文件。文件由 100 字节的关键字组成，有 10 字节的记录（这部分需要排序）。

也给出一个称为 check 的工具，看文件是否被排序，按如下格式使用：

```
check <filename>
```

基本的一次性排序按下面的过程进行：读数据，排序，写数据。但是，有多种优化方法可用：读与排序重叠进行，区分剩余记录的关键字以实现更好的 Cache 性能和更快的排序，写与排序重叠进行，等等。参见 Neuberg 等[1994]。

有一个重要的规则是起始数据必须总是在磁盘上（而不在文件系统缓存中）。实现这一规则最简单的方法是先卸载文件系统，然后重载。

你的目标是打破 Datamation 的排序记录。当前此记录是 100 万条 100 字节的数据排序时间为 0.44 秒，这个数据是在有 32 个节点的集群上获得的。如果考虑周到，你也能在配置多个磁盘的单个 PC 上打破此记录。

附录 A 流水线：基础和中级概念

事实上这是一个三段论问题。

亚瑟·柯南道尔《福尔摩斯探案集》

A.1 介绍

在阅读本章之前，许多读者可能通过其他书籍（例如我们写的更加基础的 *Computer Organization and Design*）对流水线问题有了一定的了解。由于这部分内容是第2章和第3章内容的基础，因此，要求读者在阅读本书之前对本附录的内容有比较全面的了解。尤其是在阅读第2章时更是如此。

我们从流水线的基本内容开始，包括对数据路径的讨论、对冒险的介绍以及对流水线性能的讨论。本节介绍基本的5级RISC流水线技术，该技术是本附录其他部分的基础。A.2节介绍冒险的问题，包括冒险如何引发性能问题，以及如何处理冲冒险。A.3节介绍简单的5级流水线是如何实现的，主要讨论控制问题及冒险的处理方法。

A.4节讨论指令系统设计的不同方面和流水线技术之间的交互，包括相当重要的异常处理问题，以及它与流水线的交互作用。因为A.4节是深入理解第2章内容的关键，所以对中断处理及中断处理之后的恢复过程不是很熟悉的读者更需重点关注。

A.5节讨论基本的5级流水线如何被扩展为可以处理浮点运算的长流水线结构。A.6节把上述概念结合在一起，以一个实际的超长流水线处理器——MIPS R4000/4400为例进行分析，该处理器包括有8级定点流水线和浮点流水线。

A.7节介绍动态调度的概念，以及用记分板实现动态调度的方法。由于这部分内容是对第2章核心内容（主要讨论动态调度方法）的介绍，因此将在本附录中作为相关内容进行讨论。A.7节还对第2章提到的更复杂的Tomasulo算法给出了简单介绍。尽管不涉及记分板方法也可以理解Tomasulo算法，但记分板算法更简单，也更容易理解。

什么是流水线

流水线是利用执行指令所需的操作之间的并行性，实现多条指令重叠执行的一种技术。目前，流水线已经成为了高速CPU中采用的关键技术。

流水线就像装配线那样。在汽车装配线上，一辆汽车的装配过程分为很多步骤，每一个步骤完成汽车生产的一部分。在流水线中的每一个步骤与其他任何一个步骤都并行执行，但装配的是不同的汽车。在计算机的流水线中，流水线的每一个步骤完成一条指令的一部分。就像装配线那样，不同的步骤并行完成流水线中不同指令的不同部分。每一个步骤称为一个**流水节拍**或一个**流水段**。一个流水段与另一个相连接形成流水线——指令从一端进入，经过这些流水段的处理，从另一端流出，就像汽车在装配线上的处理过程一样。

在汽车装配线上，**吞吐量**定义为每小时的汽车产量，它取决于汽车从装配线流出的速度。与之相似，指令流水线的吞吐量取决于指令流出流水线的速度。由于指令步紧密相连，所有的流水段也必须同时工作，就像装配线那样。指令沿流水线移动一次的时间间隔就是一个**机器周期**。因为所有

节拍同时工作,所以,机器周期取决于最慢的流水段,这与装配线类似,在装配线中推动装配线前进的时间间隔取决于最长的节拍。在计算机中,这个机器周期通常是一个时钟周期(有时是两个时钟周期,很少是多个)。

流水线设计者的目标是平衡各个流水段的长度,就像装配线设计者力争平衡每一步的时间那样。如果每一步都得到了最佳的平衡,那么每条指令在流水线上的平均时间在理想情况下等于

$$\frac{\text{非流水线机器1条指令的时间}}{\text{流水线机器段数}}$$

在这种情况下,流水线的加速比等于流水线的段数,就好像在一个有 n 个装配段的流水线上,可以同时有 n 部汽车在装配一样。但是,通常的流水线加速比和流水段之间不会有这么好的平衡,而且流水线需要一些附加的时间开销。因此,每条指令在流水线上的平均执行时间不会达到上面的最小值,尽管可以很接近。

流水线可以减少指令的平均执行时间。当从不同的角度看流水线时,这个减少量的含义也相应有所不同。可以认为是减少了每条指令的平均时钟周期数(CPI),也可以认为是减少了时钟周期的长度,还可以认为在这两个方面都减少了。如果研究对象是一台每条指令分为多个时钟周期的机器,那么流水线可以看做是减少了CPI——这是我们将采纳的基本观点。如果研究对象为每条指令执行一个长时钟周期,那么流水线减少的是机器的时钟周期。

流水线是一种在连续指令流中开发指令级并行性的技术。与某些加速技术相比(见第4章),流水线的明显长处是:它对编程者是透明的。在本附录中,我们将首先看到一个经典的5段流水线。在其他章节中,还讨论了现代处理器中采用的一些更加复杂的流水线技术。在继续介绍流水线之前,我们需要一个简单的指令系统,我们将在下面进行介绍。

RISC 指令系统基础

在本书的所有内容中,我们均使用RISC(精简指令系统计算机)系统结构或者load/store系统结构来对基本的概念进行说明,这些概念也适用于其他系统结构的处理器。在本节中,我们介绍典型的RISC系统结构的核心。在附录和本书中,我们默认的RISC系统结构是MIPS系统。在很多地方,这些概念是如此相似,所以我们不需要与特定的其他系统加以区别。RISC系统结构有以下几个关键特点,这些特点明显简化了RISC的实现:

- 所有运算使用的数据都来自寄存器,运算结果也都写入寄存器,每个寄存器的典型长度是32位或64位。
- 能够访问存储器的操作只有两种指令:从存储器中读取数据到寄存器的load指令和从寄存器向存储器中写数据的store指令。load和store指令可以对一个寄存器的一部分进行操作(例如,一个字节,16位或者32位)。
- 指令的数量比较少,所有指令的长度均相同。

这种结构使流水线的实现得到显著的简化,这就是为什么使用这种方法来设计指令系统的原因。

为了和本文的其他部分保持一致,我们使用MIPS64(MIPS的64位版本)。扩展的64位系统通常在操作符的开始或者结尾有一个D。例如,DADD表示64位的加指令,LD表示64位的load指令。

与其他的RISC指令系统一样,MIPS指令系统提供32位的寄存器,尽管0号寄存器的值总是0。大多数RISC指令系统(包括MIPS指令系统)都包含有三种类型的指令(详见附录B):

1. **ALU 指令**：这类指令使用两个寄存器或者一个寄存器和一个带符号的立即数（称为ALU立即数指令，在MIPS系统中它们有16位的偏移值），对它们进行操作，然后将结果存入第三个寄存器。典型的这类指令包括加（DADD）、减（DSUB）以及与32位和64位无关的逻辑操作（如AND和OR）。这些立即数指令使用相同的符号，并加上后缀I。在MIPS中，有带符号和不带符号的算术指令。不带符号的算术指令不会产生溢出——所以在32位和64位的模式下都是相同的。不带符号的算术指令在指令后面加字符U来表示（如DADDU, DSUBU和DADDIU）。
2. **load和store指令**：这些指令的操作数部分由一个称为基址寄存器的寄存器和一个称为位移量的立即数字段（在MIPS中为16位）构成。基址寄存器中的值与位移量的和就是操作的存储器地址，称为有效地址。以load指令为例，其第二个操作数（寄存器）给出从存储器中读取数据的目标地址。指令读取字（LD）或存储字（SD）是对整个64位的寄存器进行读取和存储。
3. **转移和跳转**：转移语句就是条件跳转。在RISC指令系统中通常有两种方法来指明转移的条件：使用一系列的条件位（有时称为“条件码”），或者通过寄存器之间或寄存器与0之间的比较。MIPS使用后者。在本附录中，我们只考虑两个寄存器之间的相等比较。在所有的RISC指令系统中，转移的目标地址都是通过在当前PC值上加一个位移量（在MIPS中为16位）来得到的。很多RISC指令系统中都提供了直接跳转指令，但是在本附录中我们不对这种方式进行讨论。

一个RISC指令系统的简单实现

为了更好地理解在流水线方式下如何实现一个RISC指令系统，我们首先需要理解在非流水线情况下RISC指令系统是如何实现的。本节介绍一种简单的实现方案，在这种实现方案中，执行每条指令最多只需要5个时钟周期。当把该实现方案扩展到流水线之后，CPI将大幅度降低。该方案并不是最经济和高效率的非流水线实现方案——实际上它只是用来使引入流水线概念变得更加自然。我们将在本章稍后说明如何改进该方案。在实现指令系统时需要一些并不属于这个系统结构的临时寄存器，我们引入这些寄存器以简化流水线。我们的实现方案只关注RISC系统结构的一个定点子集，包括load/store指令、转移指令和定点ALU指令。

每一条RISC指令的执行最多需要5个时钟周期。这5个时钟周期如下：

1. 取指令周期（IF）

根据PC（程序计数器）指示的地址从存储器中取指令并装入到指令寄存器（IR）中，同时PC加4（因为每条指令是4个字节）以获取下一条指令的地址。

2. 指令译码/读寄存器周期（ID）

对指令进行译码并访问寄存器堆以读出寄存器中的内容。对寄存器中的内容进行比较，判断是否是转移指令。如果需要对位移量进行处理，则把增量之后的PC值与带符号的位移量相加，得到可能的转移目标地址。在更先进的实现中（我们将在稍后进行讨论），如果转移的判断结果为“真”，则把转移处理的目标地址写入PC，这样可以在本段结束时完成对转移的处理。由于RISC指令系统制定的寄存器位置是固定的，使得译码过程和读指令过程可以同时进行。这种技术称为固定字段译码。注意，我们可能会读取一个并没有使用的寄存器，这不会对指令的执行产生任何影响（但是，这种读取确实浪费了能量，在对功耗要求比较严格的设计中，要尽量避免发生这种问题）。由于指令的立即数部分也保存在相同的位置，所以需要扩展立即数的操作也可以在本时钟周期内进行。

3. 执行/有效地址周期 (EX)

ALU 指令对上一个时钟周期准备好的操作数进行运算, 根据指令的类型执行下面三个功能中的一个。

- 访问存储器: 通过 ALU 对基址寄存器和位移量进行加法运算形成有效地址。
- 寄存器-寄存器 ALU 指令: ALU 根据操作码对从寄存器堆中读取的数据进行运算。
- 寄存器-立即数 ALU 指令: ALU 根据操作码对从寄存器堆中读取的第一个操作数和扩展后的立即数进行运算。

在 load-store 系统结构中, 因为没有指令需要在计算数据地址和指令目标地址的同时对操作数进行运算, 所以有效地址周期和指令执行周期可以合并在同一个时钟周期内。

4. 访问存储器 (MEM)

如果是 load 指令, 将根据前一个周期计算得到的有效地址从存储器中读取数据。如果是 store 指令, 则根据有效地址将第二个寄存器中的数据写入存储器中。

5. 写回周期 (WB)

- 寄存器-寄存器 ALU 指令或 load 指令。

将结果写入寄存器堆。结果可能来自存储器 (对于 load 指令) 或者来自 ALU (对于 ALU 指令)。

在这种实现方法中, 转移指令需要 2 个周期, store 指令需要 4 个周期, 而所有其他的指令需要 5 个周期。假定转移的频率为 12%, store 频率为 10%, 那么总的 CPI 为 4.54。这种实现方法无论从获取高性能方面, 还是从在相同性能条件下使用最少硬件最少方面考虑, 都不是最优的。我们把改进这种设计的任务作为练习留给读者自己来完成。

经典的 5 段流水线 RISC 处理器

我们只需要简单地在每一个时钟周期启动一条新的指令, 就可以使上面所描述的指令执行过程变为流水线执行 (由此可以看到我们为什么选择这个实现方案)。前一节中的每一个时钟周期就成了一个流水段, 即流水线的的一个周期, 如图 A.1 所示的指令执行模式, 在这个图中使用了流水线结构的典型画法。其中, 每一条指令经过 5 个时钟周期执行完成, 在每一个时钟周期内, 硬件将启动一条新的指令并执行 5 条不同指令的某个阶段。

	时钟								
	1	2	3	4	5	6	7	8	9
指令 i	IF	ID	EX	MEM	WB				
指令 $i+1$		IF	ID	EX	MEM	WB			
指令 $i+2$			IF	ID	EX	MEM	WB		
指令 $i+3$				IF	ID	EX	MEM	WB	
指令 $i+4$					IF	ID	EX	MEM	WB

图 A.1 简单 RISC 流水线。每一个时钟周期都有一条新的指令取进来并开始长达 5 个时钟周期的执行过程。若在每一个时钟周期都启动一条新的指令, 那么性能将是不进行流水处理的机器的 5 倍。流水线每一个段的名称与前面讲的非流水线的处理周期的名称相同: IF 表示取指令, ID 表示指令译码, EX 表示执行指令, MEM 表示存储器访问, WB 表示写回结果

LD R2 (00R1) IF P ALU → MEM → WB
 PL → ALU

也许读者难以理解流水线竟然会如此简单，实际上它并不简单。在这一节和后面几节里，我们将讨论一些因为流水而引入的问题，从中可以对“真实的”RISC 流水线有一个更清晰的认识。

开始，我们需要确定处理器在每一个时钟周期都进行什么样的动作，并保证在同一个时钟周期没有两条指令使用相同的数据通路资源。例如，一个 ALU 不能同时用于计算有效地址和做减法运算。因此，我们必须保证流水线中指令的重叠不会导致这样的冲突。幸运的是，由于 RISC 指令系统比较简单，因此，分析资源占用情况比较容易。图 A.2 给出了流水线方式下简化的 RISC 数据通路。可以看到，主要的功能单元都在不同的时钟周期使用，因此多条指令的重叠执行引起的冲突很少。这从下面三点可以看出。

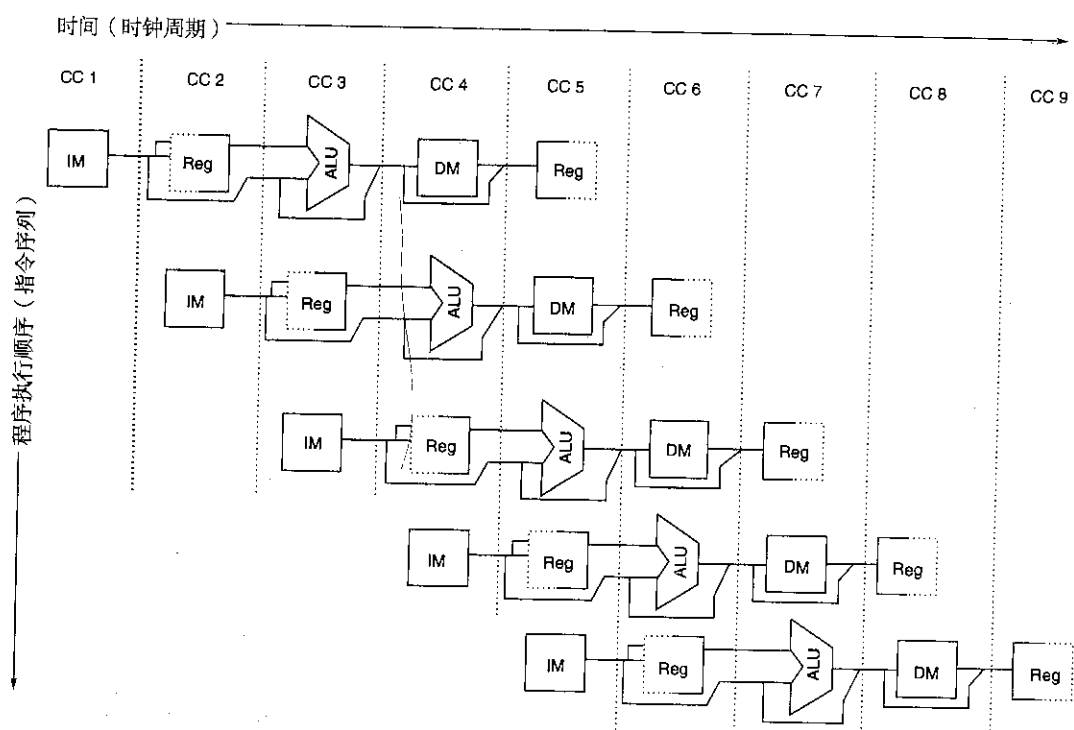


图 A.2 流水线可以看做是随时间移动的一系列数据通路。这张图显示了不同数据通路的重叠，其中周期 5 (CC 5) 表示稳定状态。由于寄存器堆在 EX 段被读，在 WB 段被写，它就出现了两次。在代表对应操作的方框中，如果是读操作 (EX)，则方框左侧画虚线，右侧画实线；如果是写操作 (ID)，则方框左侧画实线，右侧画虚线。缩写 IM 表示指令存储器，DM 表示数据存储器，CC 表示时钟周期

首先，上一节给出的基本数据通路已经使用了分开的指令和数据存储器，其典型的实现方式是使用分开的指令和数据 Cache (在第 5 章中讨论)。使用独立的 Cache 避免了对单一存储器进行取指令 and 访问数据操作之间的冲突。应该注意的是，如果我们的流水线机器的时钟周期和没有流水的机器相同，存储系统的带宽必须是原先的 5 倍，这就是取得高性能的一个代价。

其次，寄存器堆在两个流水段中被使用：ID 段读，WB 段写。这两种使用是截然不同的，因此我们简单地在两个地方画出寄存器堆。这确实意味着在一个时钟周期需要执行两次读和一次写。为了处理对同一个寄存器的读和写 (这样处理的其他原因在不久就会看到)，我们在一个时钟周期的前半部分进行写寄存器操作，在该时钟周期的后半部分进行读寄存器操作。

最后,图 A.2 并没有涉及到 PC。为了在每一个时钟周期都能够启动一条新的指令,需要每个周期都对 PC 进行自加运算并写回结果,这项工作必须在 IF 段完成,以便为下一条指令做好准备。此外在 ID 段,我们还要为可能的转移目标地址的计算提供一个加法器。一个需要进一步考虑的问题是,转移指令直到 ID 段才对 PC 寄存器的值进行改写,这种变化引入了如何来处理转移的问题。这里,我们先忽略它,以后再讨论这个问题。

尽管确保流水线中的指令在同一时间不会使用相同的硬件资源是非常重要的,但是我们还需要确保在流水线不同段中的指令不会相互影响。我们通过在连续的流水段中引入流水线寄存器来解决这个问题。在每个时钟周期结束之后,该段的所有执行结果都保存在流水线寄存器中,在下一个时钟周期作为下一个段的输入。图 A.3 是添加了流水线寄存器的流水线结构图。

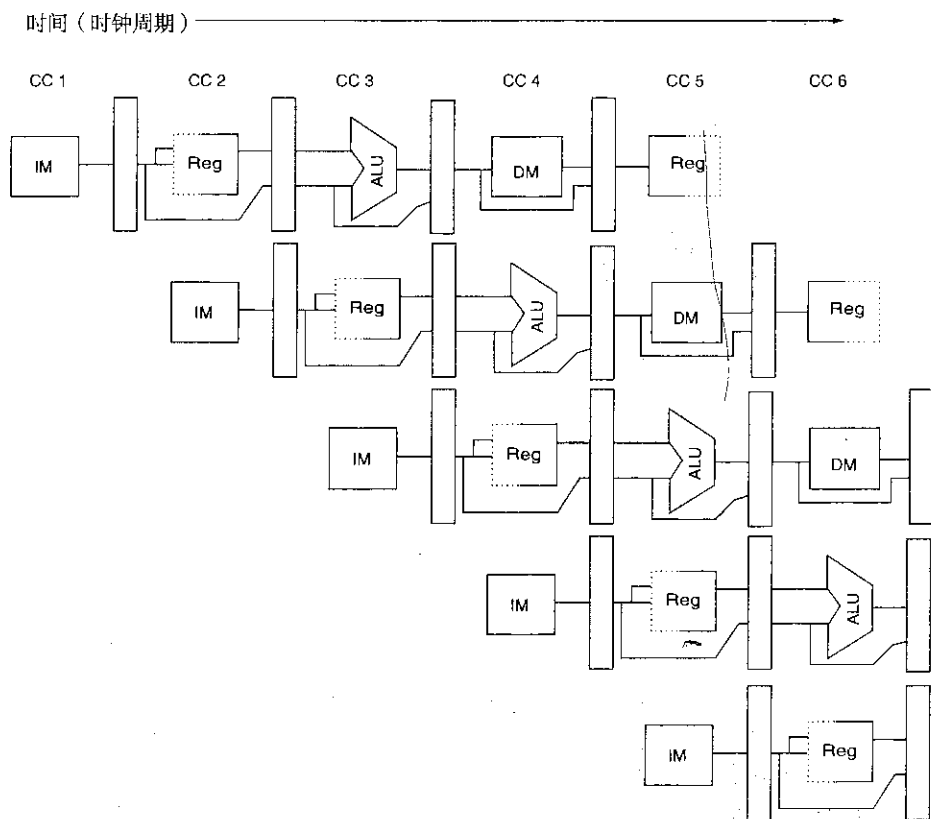


图 A.3 通过在相邻流水段之间增加一系列寄存器来进行流水。这些寄存器用于在一个流水段和下一个流水段之间进行数据传递和信息控制。这些寄存器还起到了把指令执行过程中一个阶段产生的数据传递给另一个阶段的重要作用。

尽管很多流水线结构图中为了简单起见省略了这些流水线寄存器,但是它们对于确保流水线的正常工作,它们是必需的。当然,类似的寄存器在没有流水线的多循环数据通路中也是必要的(因为只有寄存器中的数据可以在时钟的跳变过程中被保存下来)。在流水线处理器的例子中,寄存器也扮演了重要的角色,流水线寄存器把一个流水段的结果传递给需要这个数据的下一级流水线。例如,store 指令所要存储的寄存器数据值是在 ID 段被读取的,但是直到 MEM 段才被实际使用。在 MEM 段,该数据是经过两个流水线寄存器才到达数据存储器中的。同样地,ALU 指令是在 EX 段进行计算的,但是直到 WB 段才被真正保存,该结果数据也是经过两个流水线寄存器才到达数据

store(R1, R2)

IF → ID → EX → MEM → WB
R1 → 40
R2 → 40

存储器中的。有些时候，为寄存器命名是很有用的。按照惯例，用和寄存器相连的流水段来命名它们，例如 IF/ID, ID/EX, EX/MEM 和 MEM/WB。

流水线的基本性能问题

流水线增大了CPU的指令吞吐量——即单位时间完成的指令条数，但是它未减少指令各自的执行时间。实际上，流水线技术经常要对流水线附加一些控制，因而增加了开销，使单条指令执行时间略有增加。吞吐量的增大意味着程序运行得更快，总的执行时间变短，尽管没有任何一条指令的执行变快！

由于流水线没有真正减少每条指令的执行时间，这就限制了流水线的深度，关于这个问题，我们将在下一节中讨论。除了流水线时延引起的限制，流水段的不平衡和流水线的附加开销也引入了某些限制。流水段的不平衡引起的限制，是因为时钟不能快于最慢的流水段。流水线的附加开销引起的限制是因为流水线寄存器的延迟和时钟偏移。流水线寄存器增加了时钟周期的启动时间和传输延迟，其中启动时间是指从寄存器输入稳定开始，直到触发写操作的时钟信号到达为止的时间之差。时钟偏移是指在时钟到达时任何两个寄存器之间的延迟，而时钟偏移的存在也导致了对时钟周期的限制。一旦时钟周期很小，以至于与时钟偏移和锁存器附加开销相当时，流水就没有用处了，因为在一个时钟周期内没有足够时间用于有效的工作了。对此问题感兴趣的读者可以参考[Kunkel 和 Smith 1986]。就像我们在第3章中看到的那样，这种系统开销影响了 Pentium 4 相对 Pentium III 的性能提高。

例题 让我们考察上一节的非流水线型机器。假设它的时钟周期是 1 ns，ALU 操作和转移操作需要 4 个时钟周期，存储器操作需要 5 个时钟周期。以上操作的比例相应为 40%，20% 和 40%。

假设由于存在时钟偏移和启动时间，时钟周期增加了 0.2 ns，并忽略时延的影响，那么该流水线的加速比是多少？

解答：在非流水线的机器上，指令平均执行时间是

$$\begin{aligned} \text{指令平均执行时间} &= \text{时钟周期} \times \text{平均 CPI} \\ &= 1 \text{ ns} \times ((40\% + 20\%) \times 4 + 40\% \times 5) \\ &= 1 \text{ ns} \times 4.4 \\ &= 4.4 \text{ ns} \end{aligned}$$

在流水线方式下，时钟周期变慢为 $1 + 0.2$ ，即 1.2 ns，这就是指令平均执行时间。于是加速比

$$\begin{aligned} \text{流水线加速比} &= \frac{\text{非流水线指令平均执行时间}}{\text{流水线指令平均执行时间}} \\ &= \frac{4.4 \text{ ns}}{1.2 \text{ ns}} = 3.7 \end{aligned}$$

0.2 ns 的附加开销对流水线的效率附加了限制。如果不同时钟周期下附加的开销相同，那么使 Amdahl 定律就可以了解附加开销限制了加速比不能无限增大。

如果在流水线中执行的每条指令之间都是互不相关的，那么这个简单的 RISC 指令系统可以很该流水线结构中执行。但流水线中的指令很可能是相关的，这在下一节讨论。

A.2 流水线的主要障碍——流水线冒险

有一些称为“冒险”的情形,它使得指令流中下一条指令无法在设计的时钟周期内执行,这些冒险将会降低流水线可能获得的理想性能。有三类冒险:

1. **结构冒险**:也叫结构冲突,当硬件在指令重叠执行中不能支持指令所有可能的组合时发生资源冒险。
2. **数据冒险**:在同时执行的几条指令中,一条指令依赖于前一条指令的数据却得不到时,发生的冒险。
3. **控制冒险**:流水线中的转移指令或其他改写PC的指令造成的冒险。

流水线中的“冒险”可能引起流水线停顿。要消除冒险就要求流水线中的部分指令正常处理,而部分指令可以被延迟执行。对本章中讨论的流水线,当一条指令被停顿后,流水线中所有该指令之后的指令都会被停顿。流水线中该指令之前的指令必须继续执行,否则冒险就不可能被消除。这种处理的结果就是:当发生流水线停顿时,就不能再启动新的指令。在本节我们将考察几个流水线如何被停顿的例子。别担心,它们不像听起来那么复杂。

有停顿的流水线的性能

停顿会使流水线性能比理想情况差,我们用上一节的一个公式,通过一个简单的等式来看实际情况下流水线的加速比。

$$\begin{aligned}
 \text{流水线加速比} &= \frac{\text{非流水线指令平均执行时间}}{\text{流水线指令平均执行时间}} \\
 &= \frac{\text{非流水线CPI} \times \text{非流水线时钟周期}}{\text{流水线CPI} \times \text{流水线时钟周期}} \\
 &= \frac{\text{非流水线CPI}}{\text{流水线CPI}} \times \frac{\text{非流水线时钟周期}}{\text{流水线时钟周期}}
 \end{aligned}$$

需要注意的是,流水线可以看做是减小了CPI或减小了时钟周期的长度。我们可以按照传统的比较流水线的方法来使用CPI。流水线处理器的理想CPI差不多总是1,因此流水线的实际CPI是

$$\begin{aligned}
 \text{流水线CPI} &= \text{理想CPI} + \text{每条指令的停顿周期数} \\
 &= 1 + \text{每条指令的停顿周期数}
 \end{aligned}$$

如果忽略流水线的附加开销,并假设流水段平衡得很好,则两个处理器的时钟周期可以相等,因此

$$\text{加速比} = \frac{\text{非流水线CPI}}{1 + \text{每条指令的流水线停顿周期数}}$$

一种简单的情况是所有指令执行时需要的时钟周期数都相等,于是我们必须平衡它们的流水段数(又称为流水线深度)。此时非流水CPI等于流水段数,因此

$$\text{加速比} = \frac{\text{流水线深度}}{1 + \text{每条指令的流水线停顿周期数}}$$

如果没有引入流水线停顿,那么流水线的性能可以随深度的增加而明显改善。

换个角度,如果我们把流水线看做是缩短了时钟周期的长度,那么可以像流水线处理器一样假设非流水线处理器的CPI是1。因此有

$$\begin{aligned}\text{流水线加速比} &= \frac{\text{非流水线CPI}}{\text{流水线CPI}} \times \frac{\text{非流水线时钟周期}}{\text{流水线时钟周期}} \\ &= \frac{\text{非流水线CPI}}{1 + \text{每条指令的流水线停顿周期数}} \times \frac{\text{非流水线时钟周期数}}{\text{流水线时钟周期数}}\end{aligned}$$

对于流水段已经很好地平衡且没有附加开销的情况,非流水线机器的时钟周期长度与流水线机器的时钟周期长度之比等于流水线深度:

$$\begin{aligned}\text{流水线加速比} &= \frac{\text{非流水线时钟周期数}}{\text{流水线深度}} \\ \text{流水线深度} &= \frac{\text{非流水线时钟周期数}}{\text{流水线时钟周期数}}\end{aligned}$$

于是有

$$\begin{aligned}\text{流水线加速比} &= \frac{1}{1 + \text{每条指令的流水线停顿周期数}} \times \frac{\text{非流水线时钟周期数}}{\text{流水线时钟周期数}} \\ &= \frac{1}{1 + \text{每条指令的流水线停顿周期数}} \times \text{流水线深度}\end{aligned}$$

因此,当没有停顿时,加速比等于流水段数,与理想情况吻合。

结构冒险

当处理器进行流水处理时,指令的重叠执行要求功能单元能够流水,而且资源重复设置,以便流水线中的指令能自由组合。如果因为资源冲突而无法使用某种指令的组合,那么该处理器就被称为是有**结构冒险**的。

最常见的结构冒险出现在部分功能单元没有充分流水的时候,此时一系列使用该单元的指令不能按照每个时钟周期前进一拍的速率流水。另一种常见的结构冒险是因为某些资源没有足够多的副本,不能满足让流水线中的若干条指令同时执行。例如,一台机器只有一个寄存器堆写入端口,但在某些情况下,流水线可能要求在一个时钟周期内写入两次,这将产生一个结构冒险。

当有一系列指令遇到这种冒险时,流水线将停顿其中的一条指令,直到所需的功能单元能够使用为止。这种停顿将把CPI从理想值1增大。

某些流水线机器的指令和数据共享同一个存储器,结果是当一条指令包含有数据引用时,它将与下一条指令的取指令冲突,如图A.4所示。为了消除这种冒险,当需要访问数据存储器时就把流水线停顿一个时钟周期。停顿通常称为**流水气泡**或者**气泡**,因为它在流过流水线的过程中只是占据了空间而不做实际有效的工作。当讨论数据冒险时我们还会看到另一种停顿。

设计者通常不是每次都画出流水线的通路,而是用另一个只有流水段名称的简化图来表示停顿,如图A.5所示。表示停顿的方法是当没有实际操作时就标出该周期,并简单地将指令3右移(将指令3的启动和结束均延后一个时钟周期)。流水气泡的后果就是在它流过流水线的过程中占据资源。

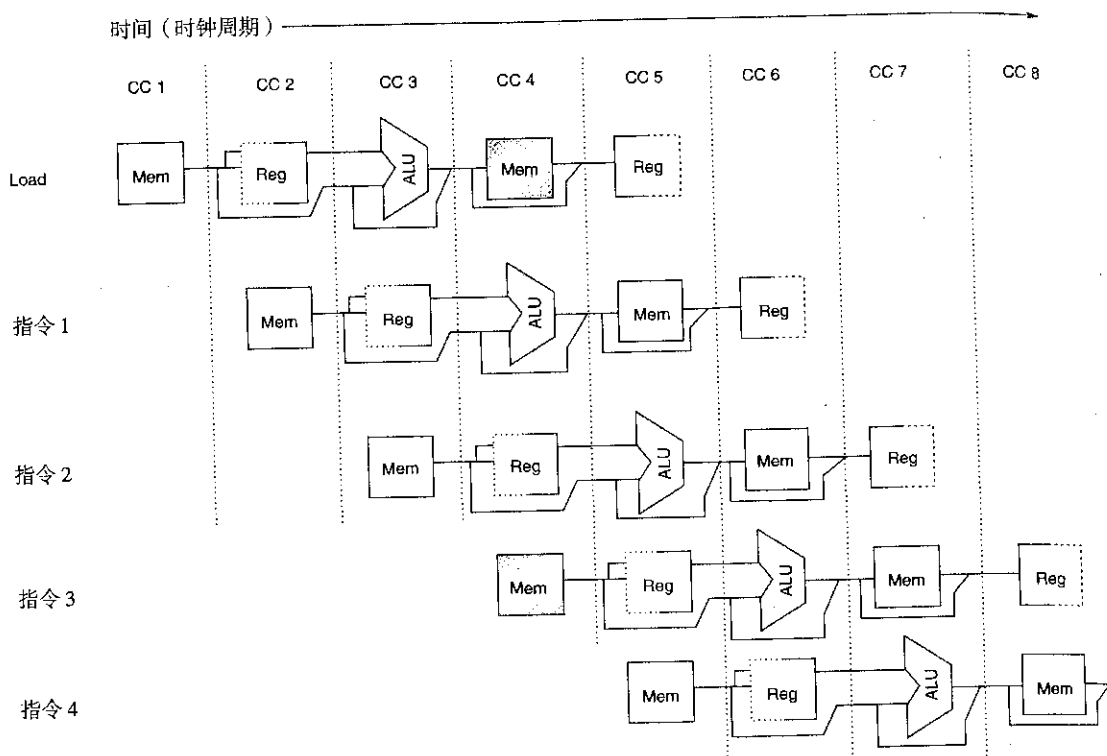


图 A.4 只有一个存储器端口的处理器,每当发生存储器引用时都会发生冲突。
在本例中 load 指令访问存储器的同时指令 3 正要从存储器中取指令

指令	时钟									
	1	2	3	4	5	6	7	8	9	10
load 指令	IF	ID	EX	MEM	WB					
指令 $i+1$		IF	ID	EX	MEM	WB				
指令 $i+2$			IF	ID	EX	MEM	WB			
指令 $i+3$				stall	IF	ID	EX	MEM	WB	
指令 $i+4$						IF	ID	EX	MEM	WB
指令 $i+5$							IF	ID	EX	MEM
指令 $i+6$								IF	ID	EX

图 A.5 单端口存储器的 load 操作引起结构冒险时的流水线停顿。load 指令窃取了一个取指令周期,于是流水线被停顿。在时钟周期 4 没有指令进入流水线(正常情况下应该是指令 3 进入流水线)。因为读入的指令被停顿,所有该指令之前的指令都能够正常处理。停顿周期将流过流水线,结果是在时钟周期 8 没有任何指令执行完成。有时,把这种流水线图表画成“停顿”占据一整行,指令 3 移到下一行。两种情况的效果是一样的,因为指令 3 到时钟周期 5 才开始执行。我们采用上面的图只是为了节省空间

例题 让我们看一下 load 结构冒险造成的影响有多大。假设数据引用占指令的 40%,在忽略结构冒险的前提下流水线的理想 CPI 是 1。若有结构冒险的处理器时钟频率是无结构冒险的处理器时钟频率的 1.05 倍,假设没有别的性能损耗,那么两种流水线哪个更快?快多少?

解答：该题有多种解法，大概最简单的方法是计算两种机器的指令平均执行时间：

$$\text{指令平均执行时间} = \text{CPI} \times \text{时钟周期长度}$$

由于没有停顿，理想情况下的平均指令执行时间就是时钟周期长度。有结构冒险的处理器指令平均执行时间是

$$\begin{aligned} \text{指令平均执行时间} &= \text{CPI} \times \text{时钟周期长度} \\ &= (1 + 0.4 \times 1) \times \frac{\text{理想时钟周期长度}}{1.05} \\ &= 1.3 \times \text{理想时钟周期长度} \end{aligned}$$

显然，没有结构冒险的处理器更快。用指令平均执行时间来计算，没有结构冒险的机器大约要快30%。对该结构冒险的一种改进方案是为指令提供单独的存储器，具体是把Cache分为指令Cache与数据Cache，或者用一个单独的缓冲栈来保存指令，称为指令缓冲栈。这两种方法均在第5章中讨论。

当别的因素相同时，没有结构冒险的机器CPI较小。那么为什么设计人员允许结构冒险存在呢？有两个原因：降低成本或者减少单元延迟。对于各种功能单元，实现流水或者增加功能单元有可能使成本过高。例如，每个周期都允许访问指令和数据Cache的机器（为了防止上例中的结构冒险）需要两倍的存储器带宽，也需要更高带宽的管脚。类似地，完全流水浮点乘法器需要很多逻辑门。如果结构冒险不经常出现，那么消除结构冒险的成本就显得太高了。

数据冒险

流水线的-一个主要影响是通过重叠执行指令来改变指令的相对执行时间。这种重叠会产生数据冒险和控制冒险，数据冒险产生的原因是流水线改变了读/写操作数的顺序，使其呈现出与在非流水线处理器上的执行时不一致的指令顺序。请看下面的指令如何在流水线中执行：

DADD	R1, R2, R3
DSUB	R4, R1, R5
AND	R6, R1, R7
OR	R8, R1, R9
XOR	R10, R1, R11

DADD指令后的所有指令都用到DADD指令的执行结果。如图A.6所示，DADD操作在WB段写R1，而DSUB操作在ID段读R1，这就称为数据冒险。除非采取预防措施，否则DSUB操作读出的将是错误的值，并错误地使用它。实际上，DSUB操作使用的值甚至是不确定的：尽管我们可能认为DSUB使用的是DADD之前R1的值，但不一定总是这样。如果在DADD与DSUB之间发生了一次中断，DADD指令的WB段就可以完成，于是R1提供给DSUB的操作就是DADD操作的结果。这种不确定性显然是难以容忍的。

AND操作也受该冒险的影响，从图A.6可见，写R1直到时钟周期5才能完成。于是，在时钟周期4读寄存器的AND操作取出的结果是错误的。

XOR操作得到了正确执行，因为它读寄存器是在周期6，此时R1已被正确写入。OR操作也不产生冒险，因为我们前半周期写寄存器堆，在后半个周期读。

下面介绍用来消除DSUB操作和AND操作数据冒险的技巧。

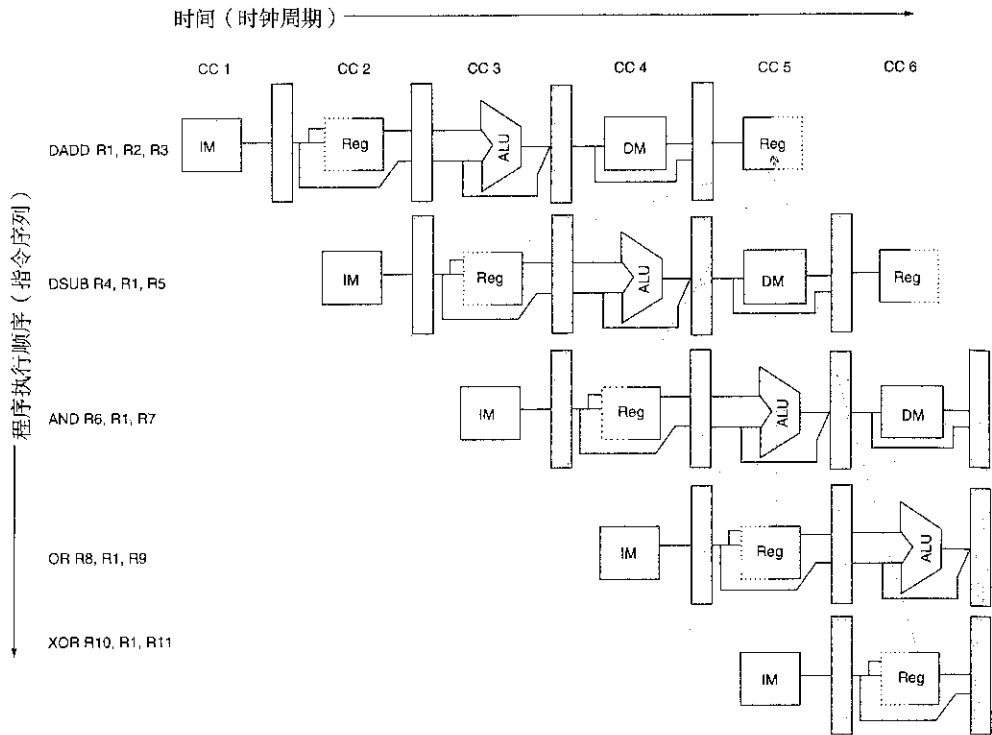


图 A.6 DADD操作的结果在后面三条指令中都使用,这就产生了数据冒险,因为寄存器在那三条指令读之后才被 DADD 操作写入

使用直通技术数据冒险引起的停顿

图 A.6 的问题可以用一个简单的硬件技术来解决,这种技术称为**直通**(也称为**旁路**,有时也称为**短路**),其关键是注意到了 DSUB 操作是在 DADD 操作产生了结果后才真正使用这个结果的。如果把 DADD 的结果从 EX/MEM 寄存器移到 DSUB 需要的地方,即 ALU 的输入锁存器,那么就没有必要引入停顿了。通过以上观察可知直通的工作流程如下:

1. 从 EX/MEM 流水线寄存器送入到 ALU 的结果总是反馈到 ALU 的输入锁存器。
2. 如果直通硬件检测到前一欠 ALU 操作写入的寄存器正好是当前 ALU 操作的操作数来源,那么控制逻辑就选择直通结果作为 ALU 的输入,而不是从寄存器堆读出源操作数。

请注意,在使用直通时,如果 DSUB 操作被停顿,那么 DADD 操作将完整执行,于是没有必要激活旁路。当这两个操作之间有一个中断时也是如此。

像图 A.6 中所示的那样,我们需要直通的操作数可能不只是来自最近的操作,而且可能来自前面三个周期启动的操作。图 A.7 就表明了这一点,并注明了寄存器读写的时刻。执行这条指令序列不需要停顿。

直通思想可以一般化,即把结果直接送到需要它的功能单元:一个结果能够从一个单元输出对应的流水线寄存器直接送到另一个单元的输入,而不限在同一单元的输出来到输入。如下面的序列:

```
DADD    R1,R2,R3
LD      R4,0(R1)
SD      R4,12(R1)
```

为了避免在该指令序列中引起的停顿,就需要把 ALU 和存储器单元的输出结果从流水线寄存器的输出直通到 ALU 和数据存储器的输入。图 A.8 画出了该示例的所有直通过路。

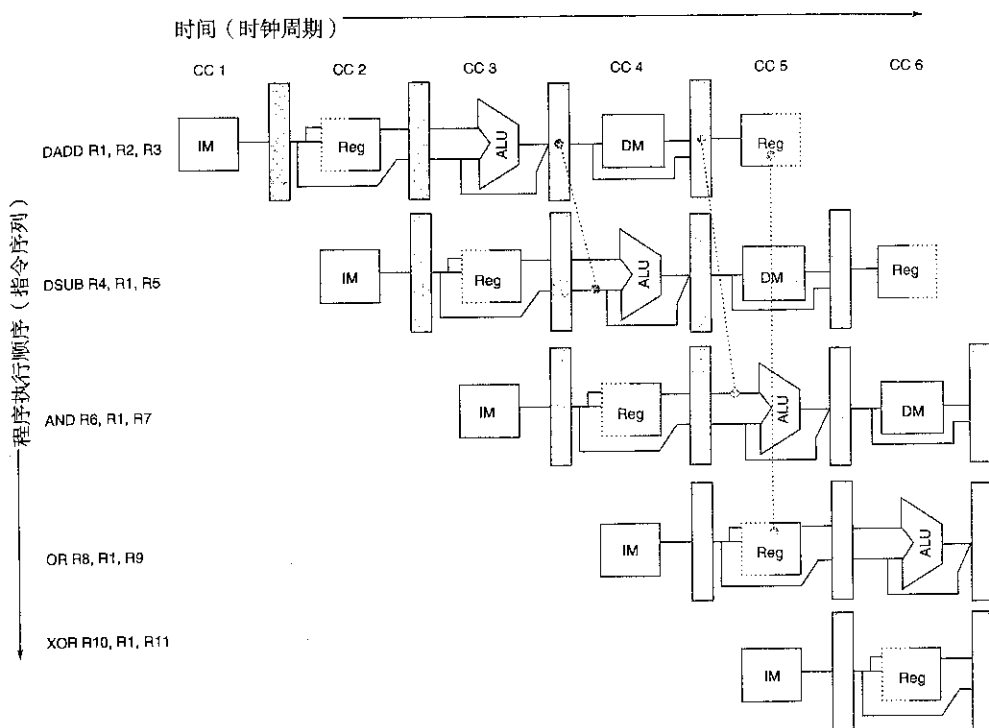


图 A.7 一些依赖 DADD 操作结果的指令使用直通来避免数据冒险。从 EX/MEM 和 MEM/WB 流水线寄存器到 DSUB 和 AND 的输入被直通到 ALU 单元的第二个输入端。OR 操作通过寄存器直通接收数据，这只需要像图中虚线所示那样对寄存器在前半个周期写、后半周期读。请注意直通结果可以从 ALU 单元的任一输入端进入，实际上 ALU 可以使用从同一个或不同流水线寄存器来的直通输入。例如，对指令 AND R6, R1, R4 就是这样

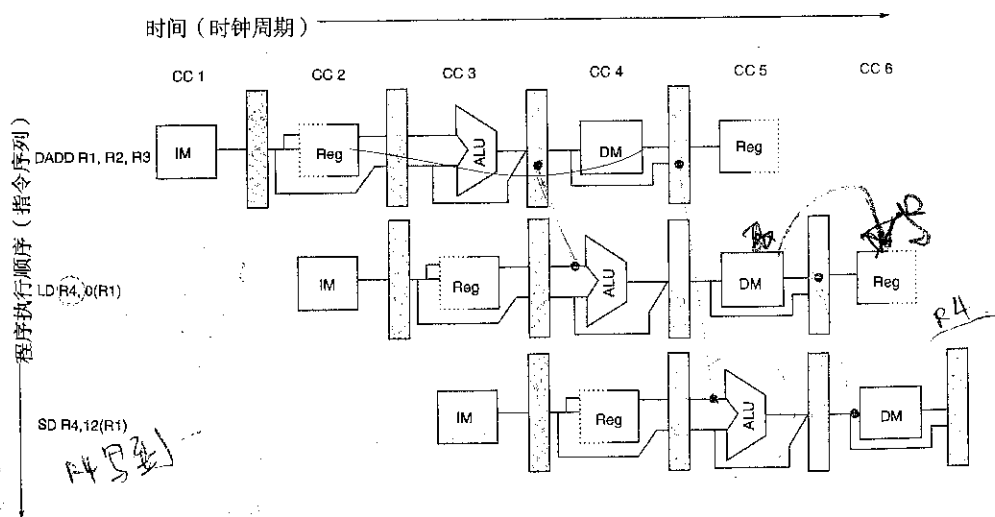


图 A.8 store 操作在 MEM 段需要一个操作数，本图画出了该操作数的直通情况。load 的结果在 MEM/WB 段从存储器输出端直通到用于保存它的存储器输入端。此外，为了计算 load 和 store 操作的地址，ALU 单元的输出端直通到它的输入端（这与把输出端连到另一个输入端并没有区别）。如果 store 依赖于前一个 ALU 操作（图中没有画），那么其值需要直通以避免停顿

需要停顿的数据冒险

遗憾的是，并非所有的数据冒险都可以采用旁路的方法来解决。请看下面的指令序列：

```
LD      R1, 0(R2)
DSUB    R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
```

本例中，带有旁路的流水线数据路径用图 A.9 表示。它与背靠背的 ALU 操作的情形不同。LD 直到时钟周期 4（它的 MEM 周期）才能得到数据，而 DSUB 在该周期开始时就需要数据。于是不能用简单的硬件消除由 load 操作造成的数据冒险。如图 A.9 所示，这种直通需要在时间上反向流动——这是计算机设计人员根本无法实现的。对于 load 启动 2 个时钟周期之后启动的 AND 操作，我们可以使它把结果立刻从流水线寄存器直通到 ALU。同理，因为 OR 从寄存器堆接收结果，它没有特殊的问题，而对 DSUB 操作，送入的数据来得太晚了，它在时钟周期开始时就需要数据，但在结束时才获得。

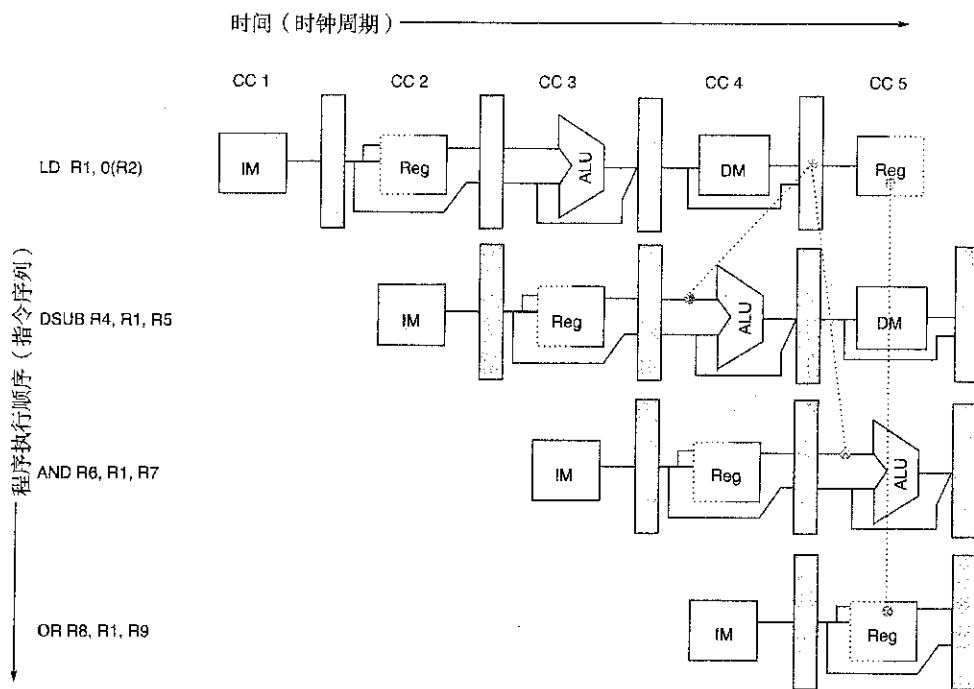


图 A.9 load 操作可以把结果直通给 AND 和 OR 操作，却不能直通给 SUB 操作，因为那意味着用“负时间”传送数据

load 操作有一种不能只用直通就能消除的延迟。这时，需要加入一种称为流水线锁定器的新硬件来保证正确运行。通常，流水线锁定器发现冒险后就停顿流水线，直至冒险被消除。在本例中，流水线锁定器停顿流水线，让需要某个结果的操作一直等到该结果产生时为止。这种流水线锁定器就像解决结构冒险那样引入了停顿。被停顿的指令其 CPI 中增加了停顿时间，在本例中是一个时钟周期。

图 A.10 使用流水段的名称来表示停顿前后的流水线。由于停顿使得 DSUB 以后的指令都后移了一个周期，因此到 AND 就变成了通过寄存器堆来直通，而 OR 根本就不需要直通。停顿插入的

流水气泡使完成该指令序列需要多用一个周期。在周期4没有启动新的操作，在周期6也就没有操作完成。

LD	R1,0(R2)	IF	ID	EX	MEM	WB			
DSUB	R4,R1,R5		IF	ID	EX	MEM	WB		
AND	R6,R1,R7			IF	ID	EX	MEM	WB	
OR	R8,R1,R9				IF	ID	EX	MEM	WB

LD	R1,0(R2)	IF	ID	EX	MEM	WB			
DSUB	R4,R1,R5		IF	ID	停顿	EX	MEM	WB	
AND	R6,R1,R7			IF	停顿	ID	EX	MEM	WB
OR	R8,R1,R9				停顿	IF	ID	EX	MEM WB

图 A.10 在上半部分可以看到为什么需要一个停顿：在相同时刻，load操作的MEM周期需要SUB操作的EX周期产生的数据。在下半部分，通过加入一个停顿解决了这个问题

转移冒险

在MIPS流水线中，控制冒险造成的损失比数据冒险更大。当一条转移指令执行时，它是否对PC是否加4是不确定的。前面我们说过，如果一条转移指令把PC改写成它的目标地址，那么该转移称为选中转移，否则称为未选中转移。如果某条指令*i*是选中转移，那么，通常要到ID段的末尾，在已经完成了地址计算和比较之后才能改变PC。

图A.11显示的是对转移指令最简单的处理方法，就是一旦在ID段发现有转移指令（在该段中进行指令译码），就对该转移指令之后的后继指令进行重新取指的操作。

转移指令	IF	ID	EX	MEM	WB		
转移后继		IF	IF	ID	EX	MEM	WB
转移后继+1				IF	ID	EX	MEM
转移后继+2					IF	ID	EX

图 A.11 转移指令在5段流水线中产生1个周期的停顿：这个周期是重复的IF段。紧跟在转移指令后面的取指令被忽略，一旦转移指令的目标地址确定了，就重新取指令。很明显，当转移未被选中时，转移后继*i*+1中的第二个IF是多余的，这将在后面重点论述

第一个IF段实际就是一个停顿，因为它没有做实际工作。如果转移未被选中，那么取指令段所得到的是正确的指令，就没有必要重新执行IF段了。稍后将讨论到如何利用这种现象，现在我们先看一下如何减少最坏情况下的转移开销。

每条转移指令造成的停顿会产生10%~30%的性能损失（取决于转移的频率）。因此，减少转移指令的开销就很重要。

减少流水线的转移代价

有许多减少因为转移延迟造成流水线停顿的方法，这里，我们介绍4种简单的编译时调度方法。在这4种方法中，对转移的处理都是静态的，即在整个程序执行过程中对每个转移的处理都一样。软件可以尽量利用硬件调度的动态特性和转移的行为来使转移开销达到最小。在第2章和第3章中，我们讨论了更高效的硬件和软件方法来对静态和动态的转移进行预测。

处理转移最简单的方法是冻结或冲刷流水线,即在转移的目标地址确定之前保存或者删除所有紧随转移的指令。这种方法的优点是硬件和软件两方面都比较简单。它就是图 A.11 所示的较早的流水线处理方法。在这种方法中,转移的开销是固定的,不可能通过软件来减少。

有一种性能更好的方法,只是略微复杂些,就是对所有转移都按未选中处理。因此必须注意在转移结果产生前不要改变机器的状态。因为不知道指令何时改变机器状态,以及怎样把变化改回去。在简单的 5 段流水线中,采用预测未选中调度策略,实现方法就是直接取下一条指令,好像转移指令只是一条普通的指令那样,流水线看起来也没有特殊之处。但是只要转移指令被选中,就需要用空操作代替取来的指令(只需清除 IF/ID 寄存器),并到目标地址重新取指令。图 A.12 说明了这一点。

未选中的转移指令	IF	ID	EX	MEM	WB				
指令 $i+1$		IF	ID	EX	MEM	WB			
指令 $i+2$			IF	ID	EX	MEM	WB		
指令 $i+3$				IF	ID	EX	MEM	WB	
指令 $i+4$					IF	ID	EX	MEM	WB
被选中的转移指令	IF	ID	EX	MEM	WB				
指令 $i+1$		IF	idle	idle	idle	idle			
转移目标			IF	ID	EX	MEM	WB		
转移目标 + 1				IF	ID	EX	MEM	WB	
转移目标 + 2					IF	ID	EX	MEM	WB

图 A.12 转移被选中(下)和转移未选中(上)两种情况下的预测未选中策略及流水线序列。当转移在 ID 段未选中时,我们已经取到了正确的指令,只需继续执行;否则,到目标地址重新取指令,并使转移之后的所有指令都停顿一个时钟周期

另一种方法是预测转移被选中,一旦完成转移指令的译码并计算出目标地址后,就假设转移被选中,到目标地址取指令。因为在我们的 5 段流水线中总是先得到转移的结果,后得到目标地址,所以这种方法不适用于我们的 5 段流水线(没有任何好处)。在某些机器中,尤其是隐含条件码和更强功能(于是也更慢)转移条件的机器中,转移的目标地址比其结果更早产生,这时,采用预测转移被选中的方法比较合适。无论是预测未选中调度策略还是预测选中调度策略,编译器都能通过组织代码来实现与硬件的最佳匹配。我们的第 4 种方法就是提供利用编译器进行优化的机会。

第 4 种方法在有些处理器中称为**转移延迟**,这种技术在早期的 RISC 处理器中被广泛采用,并且在 5 段流水线中顺理成章地使用。在被延迟的转移中,转移延迟为 1 的执行周期如下:

转移指令
 后续指令 1
 目标地址的指令(如果转移被选中)

后续指令放在**转移延迟槽**内。不管转移是否被选中,这条指令都要被流水。延迟槽长度为 1 的 5 段流水线的工作流程如图 A.13 所示。尽管延迟槽长度大于 1 是可能的,但实际上,所有带转移延迟的处理器一般只延迟一条指令,并使用其他的技术来处理有更多转移开销的情况。

编译器的任务是使后续指令可以连续执行。有许多种可用的优化方法。图 A.14 画出了对转移延迟的 3 种调度方法。

未选中的转移指令	IF	ID	EX	MEM	WB		
转移延迟指令($i+1$)		IF	ID	EX	MEM	WB	
指令 $i+2$			IF	ID	EX	MEM	WB
指令 $i+3$				IF	ID	EX	MEM
指令 $i+4$					IF	ID	EX
						IF	ID
被选中的转移指令	IF	ID	EX	MEM	WB		
转移延迟指令($i+1$)		IF	ID	EX	MEM	WB	
转移目标			IF	ID	EX	MEM	WB
转移目标+1				IF	ID	EX	MEM
转移目标+2					IF	ID	EX
						IF	ID

图 A.13 不管转移是否被选中，转移延迟操作都是一样的。延迟槽内的指令都要执行（MIPS 流水线里只有一条指令）。如果转移未选中，只需要按取来的指令执行；否则，按目标地址的指令执行。当延迟槽内的指令也是转移指令时，就产生了矛盾：如果转移未选中，就无法处理在延迟槽内的转移指令。由于这个原因，带转移延迟的机器通常不允许在延迟槽内有转移指令

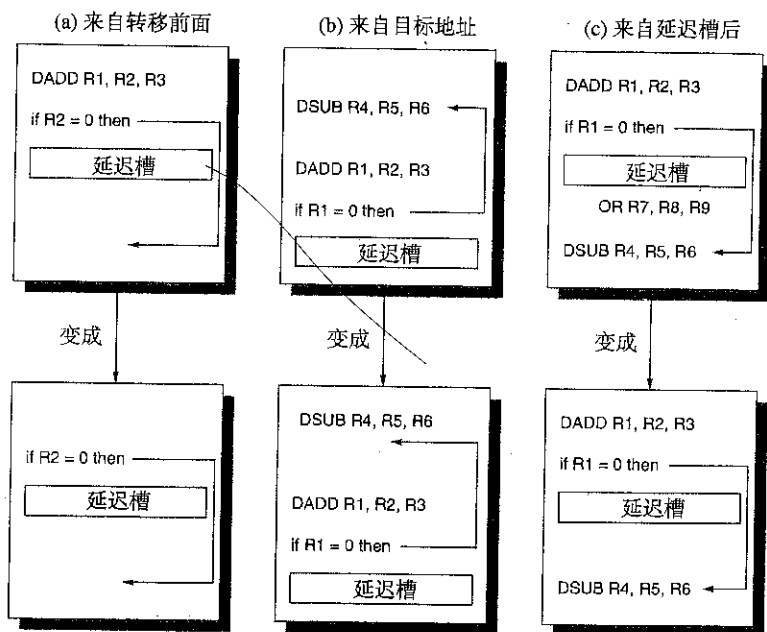


图 A.14 调度转移延迟槽。每对方框里，上面的一个是调度前的代码，下面的一个是经过调度的代码。在(a)中，填入延迟槽的是转移前的一条不相关指令，这能达到最佳效果。(b)和(c)用于(a)不可行的场合。在(b)和(c)的代码中，分支条件中使用了R1寄存器，这使DADD指令不能移到转移后面。(b)是用转移目标指令填入延迟槽，通常需要复制该指令，因为它可能被另一条指令流访问。(b)适用于转移被选中的概率较高的程序，如循环转移。(c)是用延迟槽后的指令填入延迟槽。采用(b)与(c)优化时，不管转移是否被选中，都要执行DSUB指令。因此，要求执行DSUB指令不会损坏程序的正确性，尽管有可能该指令的执行是不需要的。对于图中的例子，在转移未被选中的一段程序内，不能引用R7寄存器

采用转移延迟方法有两个限制：(1) 能够放入转移延迟槽的指令需要满足一定的条件；(2) 能否在编译时确定一个转移是否被选中的能力。我们将考察后一个限制的解决方法。为了提高编译器填充延迟槽的能力，大部分带条件转移的机器都引入了转移取消或转移无效技术。在转移取消技术中，转移指令带有对转移能否被选中的预测。当预测准确时，延迟槽中的指令像通常那样执行。否则就把延迟槽中的指令变成空操作。

转移调度的性能

这些转移调度方法各自的性能如何？假设理想情况下 CPI 为 1，那么带转移开销的流水线加速比为

$$\text{流水线加速比} = \frac{\text{流水线深度}}{1 + \text{流水线因分支造成的停顿周期数}}$$

又因为

$$\text{流水线因分支造成的停顿周期数} = \text{分支比例} \times \text{分支开销}$$

所以

$$\text{加速比} = \frac{\text{流水线深度}}{1 + \text{分支比例} \times \text{分支开销}}$$

转移比例和转移开销包括无条件转移和条件转移两个方面，但是条件转移的影响更大，因为它更经常出现。

例题 对 MIPS R4000 流水线，假设进行条件比较时没有寄存器停顿，那么在转移目标地址确定之前至少需要 3 个流水段，在判断转移条件之前还需要一个额外的周期。图 A.15 列出了三种简单的转移预测方法各自的转移开销。

请用下面的数据计算因转移开销使该流水线的 CPI 增加了多少。

	非条件转移	4%
	条件转移（未选中）	6%
	条件转移（选中）	10%

转移方法	无条件转移开销	未选中转移开销	选中转移开销
清空流水线	2	3	3
预测选中	2	3	2
预测未选中	2	0	3

图 A.15 三种简单的预测方法应用于深度流水线的转移开销

解答：计算 CPI 需要把无条件转移、未选中的条件转移和选中的条件转移与它们各自的开销相乘，结果如图 A.16 所示。

在这个长延迟流水线中，采用不同的策略，其结果有很大的差别。如果基准 CPI 为 1 并且仅有转移产生的停顿，理想流水线的速度是停顿流水线的 1.56 倍。在同样的假设下，预测未选中策略的速度是停顿流水线的 1.13 倍。

转移方法	附加到 CPI 中			
	无条件转移	未选中条件转移	选中条件转移	转移总和
发生的频率	4%	6%	10%	20%
流水线停顿	0.08	0.18	0.30	0.56
预测选中	0.08	0.18	0.20	0.46
预测未选中	0.08	0.00	0.30	0.38

图 A.16 深度流水线中三种转移预测方案的 CPI 开销

A.3 如何实现流水线

为了理解基本流水线，必须先了解简单的非流水 MIPS 的实现。

简单的 MIPS 实现方案

在本节中，我们像 A.1 节中的模式那样，首先展示一个非流水的实现方案，然后让它进行流水线化。这次我们使用的是 MIPS 系统结构。

在本小节中我们将会集中对一个包含 load-store 指令、转移指令和 ALU 指令的 MIPS 指令集进行讨论。然后，在本附录中，我们将会加入基本浮点操作。尽管我们只是对 MIPS 指令集的一个子集进行讨论，但是所有的基本规则都可以扩展到所有的指令。我们从一般的转移指令实现开始。在本节结束时我们将讨论如何实现更加强大的版本。

每一条 MIPS 指令的实现最多需要 5 个时钟周期。这 5 个时钟周期如下：

1. 取指令周期 (IF)

$IR \leftarrow Mem[PC];$
 $NPC \leftarrow PC + 4;$

操作：根据 PC 指示的地址从存储器中取指令并装入到指令寄存器 (IR)，同时 PC 加 4 以取下一条指令的地址。IR 中保存下一个时钟周期需要的指令；同样，在 NPC 寄存器中保存下一条指令的 PC。

2. 指令译码 / 读寄存器周期 (ID)

$A \leftarrow Regs[rs];$
 $B \leftarrow Regs[rt];$
 $Imm \leftarrow IR \text{ 的立即数字段进行符号扩展};$

操作：分析指令并访问寄存器堆以读寄存器 (rs 和 rt 是指定的寄存器)。通用寄存器的输出送入两个临时的寄存器 (A 和 B) 中，以供后面的时钟周期使用。IR 的低 16 位进行符号扩展并存入到临时寄存器 Imm 中，以供下一个时钟周期使用。

因为 MIPS 指令有着固定的格式 (见图 B.22)，所以，读寄存器和分析指令可以并行执行。于指令的立即数部分在 MIPS 格式中都有相同的位置，因此当下一个周期需要使用立即数时，带符号立即数的符号扩展操作也在当前时钟周期进行。

3. 执行 / 有效地址周期 (EX)

ALU 对上一个时钟周期准备好的操作数进行运算，根据 MIPS 指令的类型执行下面 4 个功能中的一个。

● 存储器引用

$ALUOutput \leftarrow A + Imm;$

操作: ALU 通过加法运算形成有效地址, 并将结果装入到寄存器 ALUOutput 中。

● register-register 指令

$ALUOutput \leftarrow A \text{ func } B;$

操作: ALU 根据操作码对寄存器 A 和寄存器 B 中的值进行运算, 结果装入到临时寄存器 ALUOutput 中。

● 寄存器 - 立即数 ALU 指令

$ALUOutput \leftarrow A \text{ op } Imm;$

操作: ALU 根据操作码对寄存器 A 存放的值和立即数 Imm 中存放的值进行运算, 结果装入到临时寄存器 ALUOutput 中。

● 转移

$ALUOutput \leftarrow NPC + (Imm \ll 2);$

$Cond \leftarrow (A == 0)$

操作: ALU 将 NPC 和 Imm 中的带符号立即数相加, 计算出分转移的目标地址。在前一个时钟周期读取的寄存器 A 中的值用于决定是否进行转移操作。由于我们只考虑一种转移操作 (BEQZ), 比较的对象是 0。注意, BEQZ 实际上是一条伪指令, 它只是把 R0 中的内容传送给 BEQ 作为一个操作数。为了简单, 我们只考虑这种转移操作。

MIPS 的 load-store 结构意味着有效地址周期和指令执行周期可以合并成一个时钟周期, 这是因为没有任何指令需要同时计算数据地址、指令目标地址和对数据进行运算。除了上面的指令外, 还有一些不同形式的跳转指令, 它们与转移指令类似。

4. 访问存储器 / 转移完成周期 (MEM)

对于所有指令, PC 都需要进行更新: $PC \leftarrow NPC;$

● 访问存储器

$LMD \leftarrow Mem[ALUOutput] \text{ or}$

$Mem[ALUOutput] \leftarrow B;$

操作: 在需要时访问存储器。如果是 load 指令, 从存储器中读出数据, 并装入 LMD (load memory data, 装入存储器数据) 寄存器; 如果是 store 指令, 则将寄存器 B 中的数据写入存储器。这两种情况下使用的地址都在上一个周期里计算, 并放在寄存器 ALUOutput 中。

● 转移

$\text{if } (cond) PC \leftarrow ALUOutput$

操作: 如果进行转移操作, PC 将被寄存器 ALUOutput 中的转移目标地址替代。

5. 写回周期 (WB)

● register-register 指令

$Regs[rd] \leftarrow ALUOutput;$

● 寄存器 - 立即数 ALU 指令

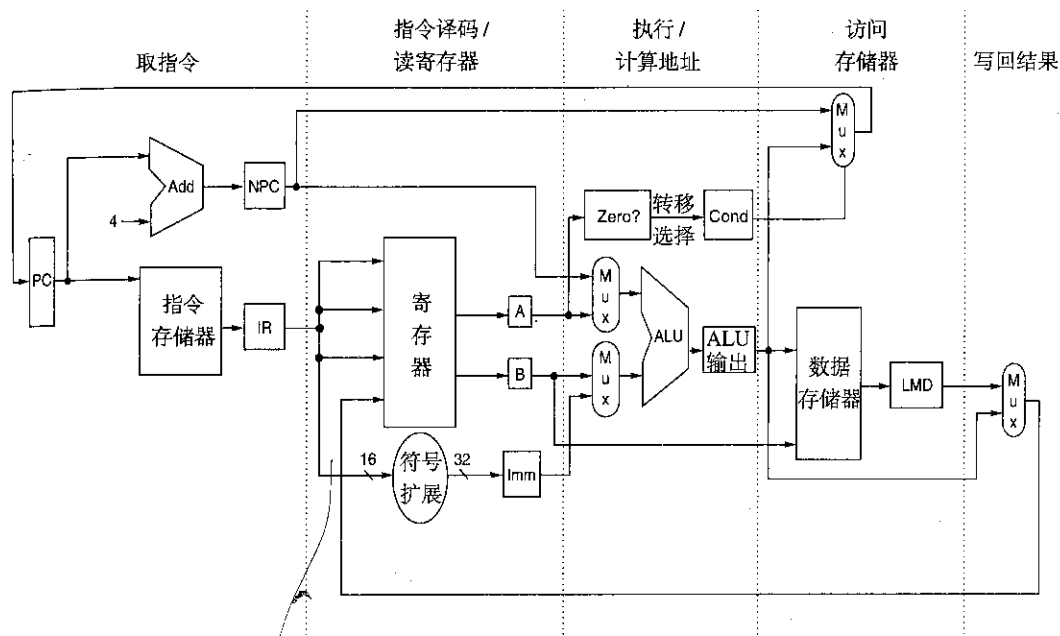
$Regs[rt] \leftarrow ALUOutput;$

● load 指令

$Regs[rt] \leftarrow LMD;$

操作：将结果写入寄存器堆，结果可能来自存储器（当数据存放在LMD中时）或者来自ALU（当数据存放在ALUOutput中时）；寄存器的写入端口在两个目标中选择一个（rd或者rt），具体选择哪一个要由操作码决定。

图A.17说明一条指令是如何沿着数据通路进行流动的。在每一个时钟周期结束时，所有在该时钟周期计算得到并在后面的时钟周期需要（不论是用于本条指令还是下一条指令）的数据将被写入到存储设备，它可能是存储器、通用寄存器、PC或者临时寄存器（如LMD，Imm，A，B，IR，NPC，ALUOutput或Cond）。临时寄存器在不同的时钟周期之间为同一条指令保存数值，而其他存储单元的状态是可见的，它们在相邻的指令之间保存数值。



图A.17 MIPS数据通路的实现，它允许在4个或5个时钟周期内完成一条指令。虽然PC在数据通路中放在取指令阶段，而寄存器在数据通路中放在指令译码/读寄存器阶段，但是这些功能单元既要被指令读取，又要被指令写入。尽管我们在这些功能单元被读取的地方标注了它们，但是PC是在访问存储器周期（和取指令周期）写入，而寄存器是在写回周期写入。对这两种寄存器，下一个流水段的写操作都由反馈给PC或寄存器数值的数据选择器（在访问存储器或写回周期）来指定。这些反馈信号使流水线复杂了很多，因为它们引入了产生冒险的可能

虽然目前所有的处理器都是用流水线技术，但是这个多周期的例子也大致反映了在早些时候大多数机器是怎样实现的。一个简单的有限状态机可以对上面5个周期的结构进行控制。对于更复杂的机器，可以使用微代码控制。在这两种情况下，上面所述的指令序列都能决定控制单元的结构。

除了降低CPI之外，这个多周期的实现方案中还可以消除一些硬件冗余。例如，这个结构中有两个ALU：一个进行PC的自加运算，另一个进行有效地址和ALU计算。由于它们并不需要在同一个时钟周期使用，我们可以通过增加一个数据选择器将它们合并，使之共享同一个ALU。与此类似，指令和数据可以保存在同一个存储器中，因为数据和指令访问发生在不同的时钟周期。

我们宁愿保留图A.17的设计而不进行优化，因为它为流水线的实现提供了一个更好的基础。除了这一节讨论的多周期设计外，我们也可以让每一条指令占用一个长的时钟周期。在这种情况下，所有的临时寄存器都可以抛弃不用，因为在一条指令的内部时钟周期之间无须通信。每一条

指令都在一个长的时钟周期内执行，并在该时钟结束时将结果写入存储器、寄存器或者PC。对于这种处理器来说，CPI将始终是1，但是其时钟周期大致等于多周期处理器的5倍，因为每一条指令都要通过所有的功能单元。设计者不会使用这种单周期的实现方法，因为有两方面的原因：首先，对于大多数CPU来说，不同的指令需要的时钟周期时间不同，单周期的实现方法效率很低。其次，单周期的实现方法需要重复的功能单元，而在多周期的实现中功能单元可以共享。不过，这个单周期的数据路径可以用来说明流水线技术是如何改善处理器的时钟周期而不是CPI的。

MIPS的基本流水线

我们只需在每一个时钟周期启动一条新的指令便可以使图A.17所示的数据通路成功流水。由于在每一个时钟周期内每个流水段是活动的，一个流水段的所有操作都必须在一个时钟周期内完成，而且任何操作组合都可能出现。另外，数据通路的流水要求从一个流水段传到下一个流水段的数据必须放在寄存器中。图A.18给出了各个流水段之间的MIPS流水线使用的寄存器，这些寄存器称为**流水线寄存器**或**流水线锁存器**。在流水线寄存器上面标有所连接的流水段。从图A.18可以清楚地看出，各个流水段之间是通过流水线寄存器相连接的。

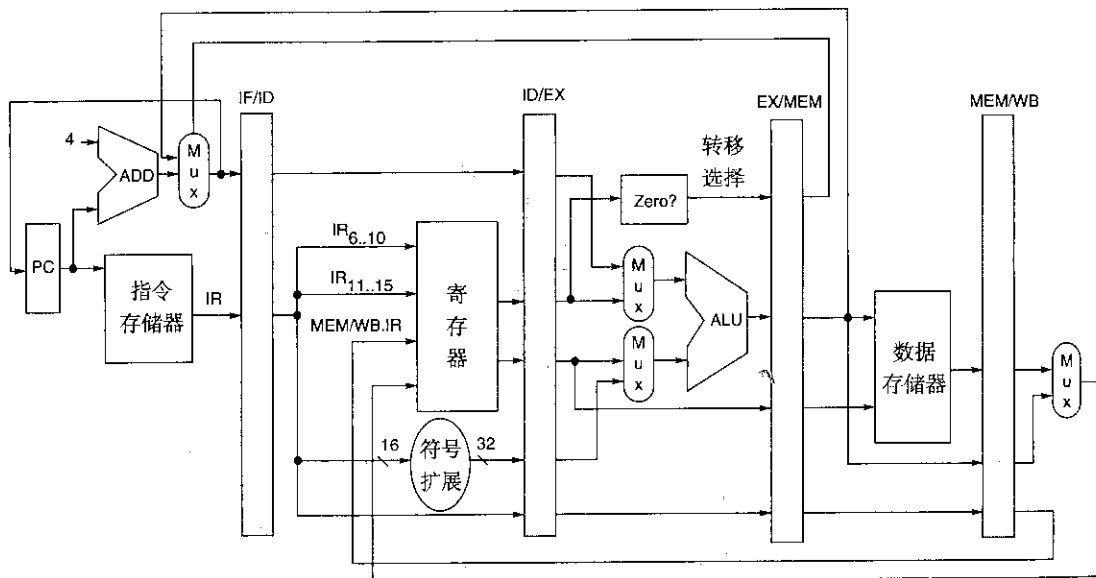


图 A.18 通过在各个流水线之间增加一系列寄存器来进行流水。这些寄存器用于在一个流水段和下一个流水段之间进行数据传递和信息控制。我们也可以将PC视为一个流水线寄存器，它位于IF之前。PC的多路开关已经经过移动使得PC能准确地在一个流水段（IF）写入。如果不移动，将会在处理转移指令时发生冲突，因为有两条指令要同时对PC写入不同的值。大多数的数据通路随时钟先后从左向右流动。从右向左的流动（载有反馈信息和转移的PC信息）使情况变得很复杂

所有用于在同一条指令的各个时钟周期之间保存临时数据的寄存器都归入流水线寄存器这一类中。属于IF/ID寄存器部分的指令寄存器（IR）的各个字段，在用于提供寄存器名时将进行标记。流水线寄存器在两个相邻流水段之间既传递数据也传递控制信息。后面流水段需要的数据，必须放在这种寄存器中，并从一个流水线寄存器复制到下一个流水线寄存器，直到不再需要它时为止。如果我们试图使用早先那种在非流水数据通路中使用的临时寄存器，数据在未使用完之前就会被覆盖掉。例如，对于load或ALU操作，写操作用到的寄存器操作数字段由MEM/WB流水线寄存器提

供而不由 IF/ID 寄存器提供。这是因为我们需要一个 load 或 ALU 操作来写由这个操作指明的寄存器，而不是当前从 IF 段转移到 ID 段的指令寄存器字段。这个目标寄存器字段仅简单地从一个流水线寄存器复制到下一个流水线寄存器，直到在 WB 段被使用。

每个时刻每条指令都只在一个流水段上是活动的，因此任何指令所做的动作都发生在一对流水线寄存器之间。因此，我们就可以根据指令的类型检查在任意一个流水段都要做些什么来看出流水线的动作，图 A.19 体现了这一观点。为了清楚地给出数据从一个流水段到下一个流水段的流动情况，流水线寄存器字段进行了命名。应该注意的是，前两个流水段的动作独立于当前的指令类型，因为指令译码只有到 ID 段结束时才进行，所以它们也肯定是独立的。IF 的动作依赖于 EX/MEM 中的指令是否是转移操作，如果是，转移的目标地址将既用于取指令，也用于计算下一个 PC；否则，由当前的 PC 完成这两个任务（我们在前面说过，转移的结果导致流水线变得更加复杂，具体情况将在后面讨论）。寄存器源操作数采用固定字段编码，这对于能够在 ID 段读取寄存器中的内容是非常关键的。

流水段	任意指令	ALU 指令	load 或 store 指令	转移指令
IF	IF/ID.IR \leftarrow Mem[PC]; IF/ID.NPC, PC \leftarrow (if ((EX/MEM.opcode == branch) & EX/MEM.cond){EX/MEM.ALUOutput} else {PC+4});			
ID	ID/EX.A \leftarrow Regs[IF/ID.IR[rs]]; ID/EX.B \leftarrow Regs[IF/ID.IR[rt]]; ID/EX.NPC \leftarrow IF/ID.NPC; ID/EX.IR \leftarrow IF/ID.IR; ID/EX.Imm \leftarrow sign-extend(IF/ID.IR[immediate field]);			
EX		EX/MEM.IR \leftarrow ID/EX.IR; EX/MEM.ALUOutput \leftarrow ID/EX.A func ID/EX.B; or EX/MEM.ALUOutput \leftarrow ID/EX.A op ID/EX.Imm;	EX/MEM.IR to ID/EX.IR EX/MEM.ALUOutput \leftarrow ID/EX.A + ID/EX.Imm; EX/MEM.B \leftarrow ID/EX.B;	EX/MEM.ALUOutput \leftarrow ID/EX.NPC + (ID/EX.Imm << 2); EX/MEM.cond \leftarrow (ID/EX.A == 0);
MEM		MEM/WB.IR \leftarrow EX/MEM.IR; MEM/WB.ALUOutput \leftarrow EX/MEM.ALUOutput;	MEM/WB.IR \leftarrow EX/MEM.IR; MEM/WB.LMD \leftarrow Mem[EX/MEM.ALUOutput]; or Mem[EX/MEM.ALUOutput] \leftarrow EX/MEM.B;	
WB		Regs[MEM/WB.IR[rd]] \leftarrow MEM/WB.ALUOutput; or Regs[MEM/WB.IR[rt]] \leftarrow MEM/WB.ALUOutput;	For load only: Regs[MEM/WB.IR[rt]] \leftarrow MEM/WB.LMD;	

图 A.19 MIPS 流水线每一个流水段发生的事件。让我们回顾一下这个流水线组织方式中，每个段所特有的流水线操作。在 IF 段，除了取指令和计算新的 PC 之外，还将增加 PC 存入流水线寄存器（NPC），以供后面计算转移目标地址时使用。这个结构和图 A.18 的结构相同，PC 在 IF 段被一个或者两个数据源更新。在 ID 段，执行读寄存器操作，对 IR 的低 16 位进行符号扩展，并沿 IR 和 NPC 传递。在 EX 段，执行 ALU 操作或者进行地址计算，并沿 IR 和 B 寄存器（如果是 store 指令）传递。同时，当指令选中转移时，将条件码的值设置为 1。在 MEM 段，更新存储器，进行转移决策，在需要时写入 PC，并传送在最后段所需的数据。最后，在 WB 段，用 ALU 的输出或者装入的值来更新寄存器。为了简化起见，我们在流水段之间传送整个 IR，尽管随着指令沿着流水线前进过程中 IR 有用的部分越来越少。

为了控制这个简单的流水线，我们只需要决定如何控制图 A.18 中数据路径的 4 个多路开关。ALU 段的两个多路开关根据指令类型来设置，它由 ID/EX 寄存器的 IR 给出。顶上的 ALU 输入多路开关根据指令是否为转移操作来决定，下面的多路开关根据指令是 register-register ALU 操作还是其他类型的操作来决定。IF 段的多路开关用于选择用当前的 PC 还是 EX/MEM.ALUOutput 的值（转移的目标地址）作为指令的地址，这个多路开关由 EX/MEM.cond 来控制。除了这 4 个多路开关之外，还有一个在图 A.18 中并未画出的多路开关，仔细看 ALU 操作的 WB 段便很清楚为什么需要这样一个多路开关。目标寄存器字段有两个来源，它依赖于指令类型是 register-register ALU 指令，还是寄存器-立即数 ALU 指令，或是 load 指令。因此，我们需要一个多路开关来选择 MEM/WB 寄存器中 IR 的目标寄存器。

MIPS 流水线控制的实现

让一条指令从译码段（ID）流动到执行段（EX）的操作通常称为发射指令，经过了这一步的指令称为已经被发射的指令。对 MIPS 定点流水线，所有的数据冒险都可以在流水线的 ID 段检查到。如果存在一个数据冒险，相应指令就在它发射前被停顿。同理，可以在 ID 段确定需要什么直通，从而再次添加适当的控制。通过较早检测锁定关系可以减少硬件的复杂性，因为除了整个流水线被停顿的情况外，硬件不需要停顿一条已经更新了机器状态的指令。另一种方法是，在一个使用操作数（在流水线的 EX 和 MEM 段）的时钟周期开始时检测到冒险或直通。为了说明这两种方法的区别，在我们所举的例子中，由 load 指令产生的数据冒险可以用检查 ID 段的方法来消除，而对 ALU 输入端的直通通路可以在 EX 段实行。图 A.20 列出了必须处理的不同情况。

情况	操作序列示例	动作
无数据相关	LD R1, 45(R2) DADD R5, R6, R7 DSUB R8, R6, R7 OR R9, R6, R7	没有冒险，因为 R1 与紧挨着下面的 3 条指令之间没有数据相关
有数据相关， 需要停顿	LD R1, 45(R2) DADD R5, R1, R7 DSUB R8, R6, R7 OR R9, R6, R7	比较器发现在 ADD 操作中使用了 R1，因此在 ADD 操作开始 EX 之前停顿 ADD 操作（包括 SUB 和 OR）
有数据相关， 通过直通消除	LD R1, 45(R2) DADD R5, R6, R7 DSUB R8, R1, R7 OR R9, R6, R7	比较器发现在 SUB 操作中使用了 R1，因此把 load 操作的结果在 SUB 操作开始 EX 之前提前送到 ALU
有数据相关， 按顺序访问	LD R1, 45(R2) DADD R5, R6, R7 DSUB R8, R6, R7 OR R9, R1, R7	不需要动作，因为 OR 操作中的读 R1 发生在 ID 的两个半段之后，而 load 操作的读数据发生在一个半段之后

图 A.20 流水线冒险检测硬件通过比较相邻指令的源操作数和目的操作数可以发现的情况。这张表说明了需要进行比较的范围，它仅限于一条写指令之后的两条指令的源操作数和目的操作数。当有停顿时，一旦继续执行，流水线的相关性将和第三种情况比较相像。当然，与 R0 有关的冒险可以被忽略，因为该寄存器总是保存 0，上面的测试可以扩展用于此处

我们首先实现 load 指令的锁定。如果有一个 load 指令引起的源操作数 RAW 冒险，那么当 load 指令处于 EX 段时，需要其数据的指令正处于 ID 段。于是我们可以用一张表来表示出所有可能的冒险，而且，通过该表就能得到一种实现方案。图 A.21 画出了当一个使用 load 结果的操作正处于 ID 段时，所有 load 操作的锁定情况。

寄存器的操作码字段 (ID/EX.IR _{0,5})	寄存器的操作码字段 (IF/ID.IR _{0,5})	匹配的操作码字段
Load	寄存器-寄存器 ALU	ID/EX.IR[rt] == IF/ID.IR[rs]
Load	寄存器-寄存器 ALU	ID/EX.IR[rt] == IF/ID.IR[rt]
Load	load, store, ALU 立即数或分支	ID/EX.IR[rt] == IF/ID.IR[rs]

图 A.21 对一条处于 ID 段的指令预测其是否需要 load 锁定，要做 3 次比较。表中的前两行检查 load 的目标寄存器是否与 ID 段中 register-register 操作的一个源寄存器相等。第 3 行检查 load 的目标寄存器是否是 load 或 store 的有效地址、ALU 的立即数或者一个条件转移。请记住，IF/ID 寄存器保存的是可能使用 load 结果的处于 ID 段的指令状态，而 ID/EX 寄存器保存的是可能成为 load 操作的处于 EX 段的指令状态

一旦检测到一个冒险，控制单元就必须在流水线中插入停顿，以防止处于 ID 或 IF 段的指令继续流动。就像我们以前讲述的那样，所有控制信息都保存在流水线寄存器中。于是当预测到一个冒险后只需要把 ID/EX 流水线寄存器的控制位（它正好是非执行码）清为全 0。此外，只需把 IF/ID 寄存器的内容循环就可以实现停顿指令。在冒险更加复杂的流水线中，也能采用同样的方法：通过比较某些流水线寄存器来预测冒险，然后改动非执行码来防止错误执行指令流。

尽管直通的情况要复杂一些，但是方法很相似。此时需要注意的是，流水线寄存器中保存的既有要直通的数据，也有源与目的寄存器字段。所有旁路通常都是从 ALU 或数据存储器输出端到 ALU 与数据存储器的输入端，或者是零检测单元。于是，实现旁路时只需要比较 IR 中处于 EX/MEM 与 MEM/WB 段的目标寄存器和处于 ID/EX 与 EX/MEM 段的源寄存器。图 A.22 说明了这些比较及可能的直通操作。

除了以上的比较和组合逻辑电路外，还需要增加 ALU 输入端的多路开关，并添加与流水线寄存器的连接。图 A.23 说明了增加 ALU 输入端的多路开关，并添加与流水线寄存器连接的相关数据通路。

MIPS 的冒险预测和直通的硬件都比较简单，我们将看到在流水线扩展到浮点后情况要复杂得多。在讨论浮点流水线之前，我们需要了解流水线中的转移处理问题。

处理流水线中的转移

MIPS 的转移指令 (BEQ 和 BNE) 要求比较两个寄存器是否相等，一般有一个是 R0。如果只考虑需要检测的 BEQZ 和 BNEZ 的情况，可以在 ID 段的末尾插入零检测，并且完成这一判断。而实现这种优化的前提是选中与未选中的两种情况下 PC 都应该尽早计算出来。想在 ID 段计算出目标地址就需要增加一个加法器，因为本来用于地址计算的 ALU 要到 EX 段才能使用。图 A.24 表示的就是修改后的流水线数据通路。借助于这个分离的加法器和在 ID 段做的转移决策，转移停顿的长度减为一个时钟周期。尽管如此，它还意味着如果 ALU 指令后面跟着一条使用这个 ALU 指令结果的转移，那么将产生一个数据冒险停顿。图 A.25 表示的是图 A.19 的流水线经过修改后的转移部分。

在某些处理器中，转移冒险比我们所举的例子要使用更多的时钟周期，因为测试转移条件和计算目标地址需要更长的时间。例如，译码与读寄存器分成不同段的处理器可能要增加转移延迟（控

制冒险的长度), 该延迟长度至少为一个时钟周期。如果不对转移延迟做处理, 它会转化成转移开销。许多老型号机器有很复杂的指令集系统, 因此有长达4个时钟周期或者更长的转移延迟; 而大型的深度流水处理器通常有6至7个时钟周期的转移开销。通常, 流水线的深度越大, 转移所需要的延迟时钟数就越多。当然, 长转移开销的机器的相对性能依赖于整个处理器的CPI。CPI较大的处理器就能够承受更长的转移开销, 因为它们的相对损失比较小。

源指令的流水寄存器	源指令的操作码	目标指令的流水寄存器	目标指令的操作码	直传结果的目的端	比较 (如果相等, 则直传)
EX/MEM	寄存器-寄存器 ALU	ID/EX	寄存器-寄存器 ALU, 立即数 ALU, load, store, branch	ALU 上输入端	EX/MEM.IR[rd] == ID/EX.IR[rs]
EX/MEM	寄存器-寄存器 ALU	ID/EX	寄存器-寄存器 ALU	ALU 下输入端	EX/MEM.IR[rd] == ID/EX.IR[rt]
MEM/WB	寄存器-寄存器 ALU	ID/EX	寄存器-寄存器 ALU, 立即数 ALU, load, store, branch	ALU 上输入端	MEM/WB.IR[rd] == ID/EX.IR[rs]
MEM/WB	寄存器-寄存器 ALU	ID/EX	寄存器-寄存器 ALU	ALU 下输入端	MEM/WB.IR[rd] == ID/EX.IR[rt]
EX/MEM	立即数 ALU	ID/EX	寄存器-寄存器 ALU, 立即数 ALU, load, store, branch	ALU 上输入端	EX/MEM.IR[rt] == ID/EX.IR[rs]
EX/MEM	立即数 ALU	ID/EX	寄存器-寄存器 ALU	ALU 下输入端	EX/MEM.IR[rt] == ID/EX.IR[rt]
MEM/WB	立即数 ALU	ID/EX	寄存器-寄存器 ALU, 立即数 ALU, load, store, branch	ALU 上输入端	MEM/WB.IR[rt] == ID/EX.IR[rs]
MEM/WB	立即数 ALU	ID/EX	寄存器-寄存器 ALU	ALU 下输入端	MEM/WB.IR[rt] == ID/EX.IR[rt]
MEM/WB	Load	ID/EX	寄存器-寄存器 ALU, 立即数 ALU, load, store, branch	ALU 上输入端	MEM/WB.IR[rt] == ID/EX.IR[rs]
MEM/WB	Load	ID/EX	寄存器-寄存器 ALU	ALU 下输入端	MEM/WB.IR[rt] == ID/EX.IR[rt]

图 A.22 直通到 ALU 两个输入端 (处于 EX 段的指令) 的数据可能来自 ALU 的运算结果 (在 EX/MEM 或 MEM/WB 段), 或来自于 MEM/WB 段的 load 指令的结果。判断是否需要直通, 要进行 10 次比较。ALU 的上下两个输入端分别对应 ALU 的两个源操作数, 这在图 A.17 和图 A.23 中已经标出了。请注意, 在 EX 段, 锁存器输出到 ID/EX, 而其源操作数来自 EX/MEM 或 MEM/WB 段的 ALU 输出端, 或是 MEM/WB 段的 LMD 输出端。这里有一个该逻辑没有涉及的复杂问题: 如何处理多条指令对同一个寄存器的写操作。例如代码队列 DADDR1, R2, R3; DADDI R1, R1, #2; DSUB R4, R3, R1, 执行逻辑必须要保证 DSUB 指令使用的是 DADDI 指令的结果, 而不是 DADD 指令的结果。通过简单地使用直通对 MEM/WB 和 EX/MEM 进行测试就可以扩展上面的执行逻辑使之可以处理这个问题。因为 DADDI 指令的执行结果在 EX/MEM 中, 而 DADD 指令的执行结果在 MEM/WB 中

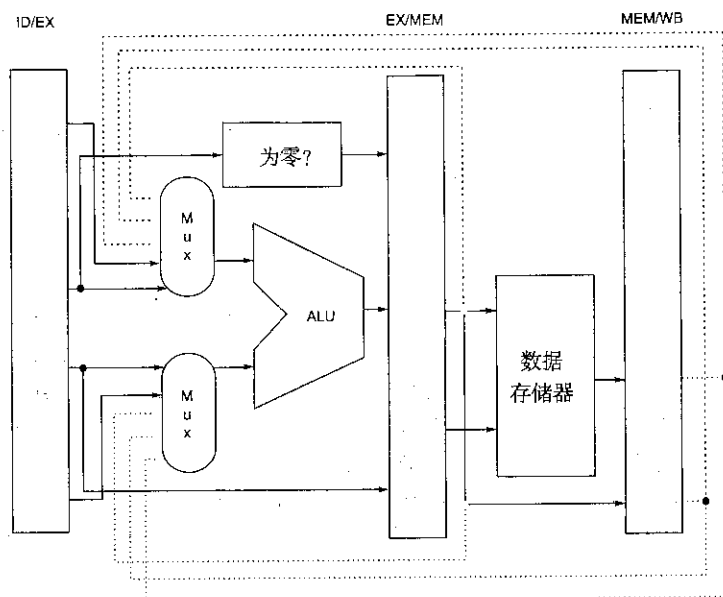


图 A.23 要直通到 ALU，需要为每个 ALU 多路开关 (Mux) 增加 3 个输入端，以及到这 3 个新输入端的相应通路。这 3 条新通路分别来自：(1) EX 段末尾的 ALU 输出端；(2) MEM 段末尾的 ALU 输出端；(3) MEM 段末尾的存储器输出端

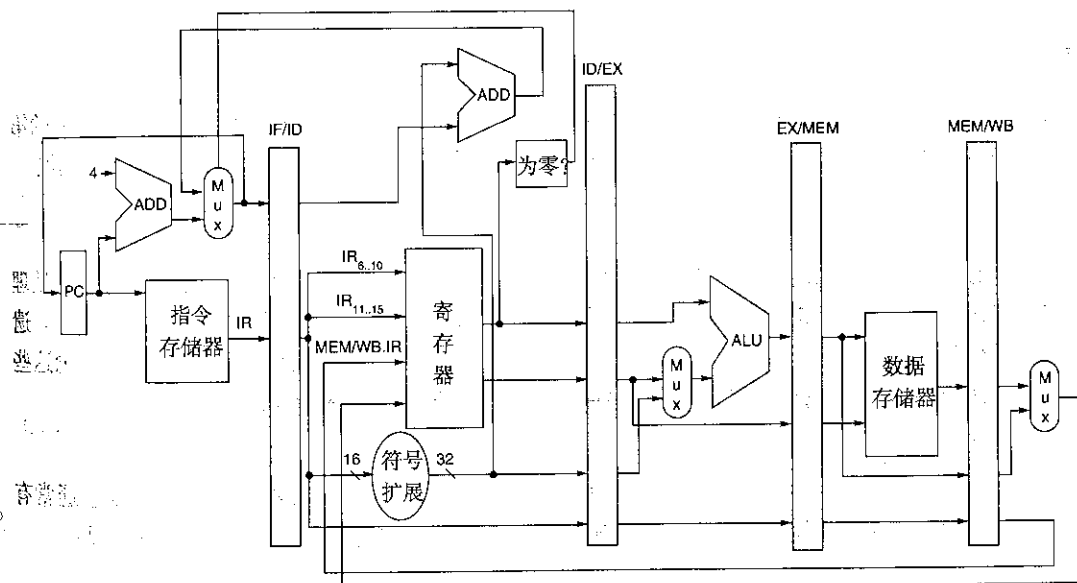


图 A.24 通过把零检测和地址计算移到 ID 段来缩短转移冒险引起的停顿。以上，我们引入了两个重要的变化，每一个变化都将原来为 3 个周期的转移停顿减少了一个周期。第一个变化是将转移目标地址的计算和转移条件的判断都移入 ID 段来完成。第二个变化是在 IF 阶段写指令的 PC，这个 PC 值可能是 ID 段计算的转移目标地址，也可能是 IF 阶段计算的 PC 增加值。作为比较，图 A.18 所示为从 EX/MEM 寄存器得到转移目标地址，在 MEM 阶段写入结果。就像图 A.18 所示的那样，PC 可以当做一个在每个 IF 段末写入下一条指令的地址寄存器

流水段	转移指令
IF	$IF/ID.IR \leftarrow Mem[PC];$ $IF/ID.NPC, PC \leftarrow (if ((IF/ID.opcode == branch) \& (Regs[IF/ID.IR_{6..10}] op 0)) \{IF/ID.NPC + sign-extended (IF/ID.IR[immediate field] \ll 2) \} else \{PC+4\});$
ID	$ID/EX.A \leftarrow Regs[IF/ID.IR_{6..10}]; ID/EX.B \leftarrow Regs[IF/ID.IR_{11..15}];$ $ID/EX.IR \leftarrow IF/ID.IR;$ $ID/EX.Imm \leftarrow (IF/ID.IR_{16})^{16} \# IF/ID.IR_{16..31}$
EX	
MEM	
WB	

图 A.25 基于前面的图 A.19 修改的流水线结构。它在 ID 段用图 A.24 所示的分离加法器计算转移的目标地址。新增加和改动过的操作都很明显。因为是在 ID 段计算转移地址, 所以对所有指令都经过这一步计算, 转移条件 ($Regs[IF/ID.IR_{6..10}] op 0$) 也对所有指令都有效。对转移后的 PC 与原序列的 PC 做选择仍在 IF 段进行, 但是现在使用的值已经不是来自 EX/MEM 寄存器的值, 而是来自 ID/EX 寄存器的值。而 ID/EX 寄存器的值是由前面的指令决定的。这个变化使得转移开销减少了 2 个周期: 提前评估转移条件和转移目标; 在同一个周期进行 PC 选择的控制。由于 cond 设为 0, 除非 ID 段是一条转移指令, 否则机器就必须在 ID 段结束前译码。又因为是在 ID 段末尾进行转移, EX, MEM 和 WB 对转移指令来说就没意义了。对于比转移指令偏移更大的跳转指令, 情况又有所不同, 我们可以用另一个附加的加法器计算 IR 左移 2 位之后得到的低 26 位与 PC 之和

A.4 实现流水线的困难

我们已经知道了如何检测及消除冒险, 现在来解决一些我们在前面暂时回避的问题。本节的第一部分讨论使指令执行顺序变得不可预测的异常情况, 第二部分讨论不同指令系统的一些问题。

处理异常

在流水线机器中处理异常情况很困难, 因为指令的重叠执行增大了判断一条指令是否改变处理器状态的难度。在流水线 CPU 中, 指令一个节拍一个节拍地执行, 几个时钟周期后才能完成。遗憾的是, 流水线中别的指令可能产生异常使这条指令在完成前被强行推出流水线。在具体讨论这些问题及其解决办法之前, 需要知道可能有哪些异常, 以及它们对系统结构的需求。

异常的种类和需求

异常是指正常的指令执行顺序被改变了的情况, 不同机器对异常有不同的描述方法。通常有中断、故障和异常等几种叫法。像在许多机器里那样, 我们把它称为异常, 包括如下情况:

- I/O 设备请求。
- 用户程序请求操作系统服务。
- 执行跟踪指令。
- 断电 (程序员的中断要求)。
- 定点运算上溢或下溢。
- 浮点运算异常。
- 页面故障 (非主存储器)。

- 访问存储器时使用错误地址。
- 存储保护违例。
- 使用未定义或未实现的指令。
- 硬件故障。
- 电源掉电。

当特指某一种具体机器异常时，通常使用比较长的名字，例如输入输出中断、浮点数异常或页面故障等。图 A.26 列出了同一种异常事件的不同名称。

异常事件	IBM 360	VAX	Motorola 680x0	Intel 80x86
I/O 设备中断	输入输出中断	设备中断	异常 (0...7 级自动向量)	向量中断
用户程序请求操作系统服务	管理程序调用中断	异常 (改变到管态方式)	异常 (指令未实现)——指 Macintosh 机	中断 (INT 指令)
执行跟踪指令	不使用	异常 (故障跟踪)	异常 (跟踪)	中断 (单步跟踪)
断点	不使用	异常 (断点故障)	异常 (非法指令或断点)	中断 (断点跟踪)
定点算术上溢或下溢；浮点陷阱	程序性中断 (上溢或下溢异常)	异常 (定点上溢陷阱或浮点下溢陷阱)	异常 (浮点协处理器错误)	中断 (上溢陷阱或算术单元异常)
页面故障 (非主存储器)	不使用 (仅 370 使用)	异常 (非有效变换故障)	异常 (存储器管理单元错误)	中断 (页面故障)
访存时使用错误地址	程序中断 (规范异常)	不使用	异常 (地址错误)	不使用
存储保护违例	程序中断 (保护异常)	异常 (访存冲突故障)	异常 (总线错误)	中断 (保护异常)
使用未定义指令	程序中断 (操作码异常)	异常 (代码优先权/保留故障)	异常 (非法指令或断点/未实现指令)	中断 (无效操作码)
硬件故障	机器检验中断	异常 (机器检验失效)	异常 (总线错误)	不使用
电源掉电	机器检验中断	紧急中断	不使用	不可屏蔽中断

图 A.26 4 种系统结构中对常用异常事件的不同叫法。在 IBM360 和 80x86 中，各种事件都称为中断，而在 680x0 中都称为异常。在 VAX 中把事件分为中断和异常，设备、软件、紧急事件称为中断，而异常又细分为故障、陷阱和失效等

尽管我们对这些事件统称为异常，但是具体事件的特点决定了硬件应该具体做什么操作。异常对硬件的需求可以有 5 种分类方法。

- 1. 同步与异步：**对相同的数据和相同的存储器分配，如果程序每次执行时，一个事件都在程序的相同点发生，这种事件称为同步事件。包含有硬件故障的异步事件是由处理器与存储器以外的设备引起的。异步事件通常是在当前指令执行完成之后的处理，因为这样做比较容易实现。
- 2. 用户请求与强制执行：**如果用户任务直接请求某个事件，那么该事件称为用户请求事件。这种事件在某种程度上不是真正的异常，因为它们是可以预测的。这类事件之所以被视为异常，只是因为它们对机器状态的保存和恢复处理与异常事件一样。因为产生这个异常的指令的唯一功能就是激活这个异常，所以用户请求异常可以在用户请求执行完成以后再处理。强制执行异常是由一些用户程序无法控制的硬件事件引起的，因为这类事件不可预测，所以它们通常比较难处理。

3. 用户可屏蔽与用户不可屏蔽: 如果用户任务可以屏蔽或禁止这个事件的发生, 那么这个事件是用户可屏蔽的。这种屏蔽只是控制硬件是否能够响应该异常事件。
4. 指令内与指令间: 这种分类方法是依据事件是否因在指令执行阶段发生而阻止指令完成(不管该事件多么短), 或者说是它是否是在指令执行期间被识别出来。指令内的异常通常都是同步的, 因为正是指令激活了异常事件, 所以每次执行都产生同样的异常。由于发生指令内异常时指令必须停下来, 所以指令内异常更难处理。指令内的异步异常是由灾难性情况(如硬件故障)引起的, 通常会造成程序终止。
5. 恢复与终止: 如果中断发生后程序的执行就停止, 那么它是一个终止事件, 否则是恢复事件。终止事件更容易处理, 因为机器不必在处理完异常事件之后重新执行原来的程序。

图 A.27 根据以上 5 种分类方法对图 A.26 中的例子进行了分类。比较困难的是对恢复事件中的指令内异常进行处理, 处理这种异常需要激活另一个程序, 并保存当前程序状态, 然后处理发生的异常, 最后恢复程序原先的状态, 继续执行当前程序。这个过程必须对当前正在执行的程序是透明的。如果一台流水线机器有这种能力, 就说这台机器是可重新启动的。以前的超级计算机和微机都缺少这种能力, 现在几乎所有机器都具备了这种能力, 至少定点流水线机器是可以重新启动的, 因为实现虚拟存储器(见第 5 章)必须使用这种技术。

异常类型	同步 与异步	用户请求 与强制执行	用户可屏蔽 与不可屏蔽	指令内 与指令间	恢复 与终止
I/O 设备请求	异步	强制执行	不可屏蔽	指令间	恢复
操作系统调用	同步	用户请求	不可屏蔽	指令间	恢复
执行跟踪指令	同步	用户请求	用户可屏蔽	指令间	恢复
断点	同步	用户请求	用户可屏蔽	指令间	恢复
定点算术溢出	同步	强制执行	用户可屏蔽	指令内	恢复
浮点算术上溢或下溢	同步	强制执行	不可屏蔽	指令内	恢复
页面故障	同步	强制执行	用户可屏蔽	指令内	恢复
访存时使用错误地址	同步	强制执行	不可屏蔽	指令内	恢复
存储保护违例	同步	强制执行	不可屏蔽	指令内	终止
使用为定义指令	同步	强制执行	不可屏蔽	指令内	终止
硬件故障	异步	强制执行	不可屏蔽	指令内	终止
电源掉电	异步	强制执行	不可屏蔽	指令内	终止

图 A.27 有 5 种分类方法来确定对图 A.26 所示的各种异常如何处理。虽然软件经常选择终止程序, 但对于再继续异常, 程序必须继续执行。发生在指令内的同步强制执行再继续异常是最难处理的

停止与重新启动执行

像非流水线机器一样, 最难处理的异常有两个特征: (1) 它在指令内(即指令执行到 EX 或 MEM 段时)发生; (2) 它必须是可重新启动的。例如我们的 MIPS 流水线中, 取数据造成的虚拟存储器页面故障仅发生于指令的 MEM 段。当页面故障发生时, 另外的几条指令正在执行。页缺失必须是可重新启动的, 它要求创建另外一个进程, 这可以由操作系统来完成。因此流水线必须安全地停下来, 并保存当前状态以便重新启动。重新启动通常是用保存重新启动点的 PC 来实现的。如果重新启动的指令不是转移指令, 则可以顺序取指令, 按通常的方式执行。否则, 需要判断转移条件, 确定在何处取指令。当异常发生时, 流水线控制器可以按以下步骤保护流水线。

1. 在下一个IF段把一条陷阱指令插入流水线。
2. 在陷阱指令进入流水线后，禁止流水线中该指令之后的所有指令的写操作，直到陷阱指令流出流水线为止。这只需对流水线内所有指令的流水线寄存器清0，其中包括产生异常的那条指令，但不包括之前的指令。这就防止了在处理完异常之前，那些未执行完的指令状态被改变。
3. 当操作系统的异常处理程序获得控制权时，它立刻保存异常指令的PC。以后，这个保存的值被用来从异常处理程序中返回。

在上一节里，当我们使用转移延迟技术时，如果只有一个PC，那么由于流水线中的指令可能不是连续执行的，因此无法恢复机器的状态。所以，PC的数目要比转移延迟长度还多一个。这是由上面的第三步来完成的。

当异常处理结束后，由特殊指令为PC装入指定地址，以恢复机器状态，并重新启动指令流（在MIPS中是使用RFE指令）。如果流水线可以停下来使当前指令都能执行结束，而当前指令之后的指令都可以重新启动，那么这种流水线称为**精确异常流水线**。理想情况下，当前指令不会改变机器状态，因为正确处理异常就要求当前指令对机器状态没有影响。对某些异常，例如浮点异常，某些机器的当前指令在进行异常处理之前就写入了它的运算结果。此时，硬件必须能够恢复源操作数，即使一个源操作数与目标操作数相同。由于浮点运算需要许多个周期，在此期间，流水线中的其他指令可能已经改写了源操作数（在下一节将看到，浮点运算通常是按乱序完成的）。为了避免这种情况，许多新的高性能CPU采用两种工作模式。一种模式下有精确异常，另一种模式下则没有。当然，精确模式执行得较慢。在某些高性能处理器中，如Alpha 21064，Power 2，MIPS R8000，精确模式经常要慢很多（超过10倍），所以，通常只在调试源代码时才采用精确异常模式。

支持精确异常模式是许多机器的需求，但是对另一些机器，它的价值仅仅是简化了操作系统接口。至少任何需要页式存储器或者IEEE算术陷阱处理的处理器都必须通过硬件或软件的支持来获得精确异常模式。对于定点流水线，实现精确异常比较容易，如果要求有虚拟存储器，就很需要支持精确异常模式。这就使许多系统结构设计人员在应用中为定点流水线提供精确异常。我们在本节中介绍MIPS定点流水线是如何实现精确异常的，在A.5节介绍更复杂的浮点流水线的异常处理技术。

MIPS中的异常

图A.28给出了MIPS的流水段和每个节拍可能出现“问题”的异常。对于流水线来说，由于有多条指令在同时执行，所以每个时钟周期都可能发生多个异常。例如，对于下面的指令序列：

LD	IF	ID	EX	MEM	WB	
DADD		IF	ID	EX	MEM	WB

在LD处于MEM段且DADD处于EX段的时刻，这两条指令可能同时发生数据页面故障和算术异常。对这个例子，可以只处理异常故障，然后重新启动。第二条指令还将出现异常（但是，如果软件设计没有错误，第一条指令将不再出现异常），此时可以单独处理第二条指令的异常。

实际上，发生异常的情况不像这个例子这样清晰。异常发生的顺序可能是乱序的，也就是说，可能靠后的指令比靠前的指令先出现异常。同样看上面的两条指令，可能LD指令在MEM段出现页面故障，而后面的DADD指令在IF段出现页面故障，此时就出现了顺序混乱的情况。

为方便起见，我们称靠前的LD指令为指令*i*，靠后的DADD指令为指令*i+1*。由于我们是在实现精确异常处理，就应该先处理LD指令造成的异常，不能简单地按异常出现的先后顺序进行处理，否则将与它们非流水时的结果不同。实际上，硬件为每条指令分配了一个异常状态向量，当指

令在流水线中执行时,异常状态向量也随之变化。一旦该向量被标为出现了异常,那么就关闭所有写操作(包括写寄存器和写主存储器)。例如 store 可能在 MEM 段造成异常,硬件必须在发生异常时阻止 store 完成。

流水段	可能发生的异常
IF	取指令时的页面故障;访问存储器时使用错误地址;存储保护违例
ID	未定义或非法的操作码
EX	算术异常
MEM	取数据时页面故障;访问存储器时使用错误地址;存储保护违例
WB	无

图 A.28 MIPS 流水线中可能发生的异常。由指令或数据访问存储器引起的异常占了 8 种异常中的 6 种

当一条指令进入 WB 段(或者将离开 MEM 段)时,系统就检查异常状态向量。如果向量表明该指令出现了异常,就对所有有异常的指令按照它们在流水线中的顺序处理——最先执行的指令(流水线中最早的节拍)引发的异常最先处理。这就保证了总是先处理靠前的指令 i 造成的异常,后处理靠后的指令 $i+1$ 造成的异常。当然,指令 i 靠前的段可能是非法的,但是由于已经禁止它对寄存器堆和主存储器的写操作,所以指令 i 没有改变机器的状态。在 A.5 节中,我们将看到在浮点操作中实现这种精确模式将更为困难。

在下一小节中,我们将讨论在设置有更强的功能单元、指令执行时间更长的流水线机器上,处理异常变得更加困难的情况。

指令系统引起的复杂问题

一条 MIPS 指令最多只产生一个执行结果,而 MIPS 流水线只允许在指令执行结束时写这个结果。如果一条指令能够保证执行完成,就称为已提交指令。MIPS 定点流水线中所有指令到达 MEM 段末尾(或 WB 段开始)时都是已提交的,并且指令不能在 WB 段开始前改变机器状态。因此, MIPS 处理器实现精确异常处理比较简单。但是在一些处理器中,某些指令可以在执行过程中,它自己和别的指令都未被提交时就改变机器状态。例如 IA-32 的自增寻址方式就是在指令执行中间改变机器状态。在这种情况下,如果指令因为异常而中途停止,它留下的就是错误的处理器状态。尽管我们知道是哪条指令引起了异常,但是如果没有专用的硬件,我们就难以知道这条半途终止的指令都做了什么,因此重新启动指令流就变得很困难。尽管可以避免在指令结束前改写机器状态,但这可能比较困难,开销也很大,因为改变了的状态可能存在相关。例如对一条多次自增同一个寄存器的 VAX 指令。为了实现精确异常处理模式,目前,大部分带有这样的指令的机器都能够在指令被提交前恢复机器状态。在下一节中我们将看到,更强大的 MIPS 浮点流水线也有类似的问题, A.7 节将介绍很复杂的异常处理。

在指令执行过程中,改写存储器的指令还会引起一些新的问题,例如 VAX 或 IBM 360 机器中的串复制运算。为了能够中断和重新启动这些指令,必须规定它们使用通用寄存器作为工作寄存器,于是该指令的工作状态就被保存在寄存器中。当发生异常时,机器可以参照寄存器中的内容处理该指令。在 VAX 中,当一条指令改写了主存储器时,保存其状态的寄存器的一个附加位就做了记录,于是在重新启动时, CPU 就知道从这条指令的起始还是中间开始执行。IA-32 的串指令也是这样使用寄存器的。

有一些指令的状态很奇怪,可能导致新的流水线冒险或需要额外的硬件保存状态。条件码就是这样的一个例子。许多处理器是把条件作为指令的隐含部分,这样做的优点是:把对条件码的判断和真正的转移操作分离开。但是,由于大部分指令设置了条件码,却不在条件判断和转移操作之间

使用条件码，所以隐含的条件码与转移指令之间的流水线延迟和调度比较困难。此外，在带有隐含条件码的机器里，处理器还必须判断何时确定转移条件，这就要找出在转移之前条件码何时被设定。在大部分有隐含条件码的机器里，是通过使转移前的所有指令都有机会设置条件码，然后进行转移判断实现这一功能的。

当然，直接设置条件码的处理器允许对条件测试指令与转移指令之间的延迟时间进行调度，但是流水线必须跟踪最后一条设置条件码的指令以确定何时可以判断转移条件。从效果上讲，条件码必须像 RAW 转移冲突那样进行检测，这种做法与 MIPS 对寄存器的处理很相似。

最后一个难处理的问题的是多周期运算，请看下面的 VAX 指令序列：

MOVL	R1,R2	；在寄存器之间传送
ADDL3	42(R1),56(R1)+,@(R1)	；增加存储器单元
SUBL2	R2,R3	；减寄存器
MOVC3	@(R1)[R2],74(R2),R3	；传送字符串

这些指令需要的时钟周期数差别极大，有的只需要一个周期，有的需要上百个时钟周期。它们需要访问的存储器地址也相差极大，从零到几百，数据冒险很复杂，包括指令间和指令内的冒险。如果想用同样的时钟周期执行这些指令，将引入大量冒险和旁路，并需要极长的流水线，因此是不可行的。所以，在指令级对 VAX 进行流水很困难。但是 VAX 8800 的设计者找到了一个聪明的解决办法。它们对微指令进行流水。微指令是用于实现复杂指令的简单指令。由于微指令很简单（它们看起来像 MIPS 中的指令），流水线的控制也简单多了。从 1995 年开始，Intel 所有的 IA-32 微处理器已经开始使用这种技术来把 IA-32 指令转化为微指令进行流水。

相比较而言，采用 load-store 结构的处理器对同样的任务可以设计出更简单的操作序列，也更易于流水。如果系统结构设计者能够解决好指令系统设计和流水线技术之间的关系，它们就能设计出性能更好的流水线。下一节中我们将讨论 MIPS 流水线如何处理长周期指令，尤其是浮点操作指令。

在过去的很多年里，指令系统的设计和实现之间的互动很小。如何实现并不是指令系统设计者考虑的主要问题。直到 20 世纪 90 年代才明确了指令系统的复杂性会影响流水线的效率和增大实现的难度。

A.5 扩展 MIPS 流水线以处理多周期操作

现在研究如何将我们的 MIPS 流水线扩展到可以处理浮点操作。在这一节中，我们将集中讲述基本方法和可行方案，并在最后附上一个对 MIPS 浮点流水线的性能评测。

要求在一个甚至两个时钟周期中完成所有的 MIPS 浮点操作是不现实的。这样做或者意味着大大延长时钟周期，或者意味着在浮点单元中使用大量逻辑电路，甚至两者都出现。因此，我们允许浮点流水线操作具有较长的延迟时间。假如我们设想浮点指令具有和定点指令相同的流水线，这就很容易理解了。当然，要有两个重要的改动。首先，为了完成浮点操作，根据需要 EX 流水段要循环重复多次，对不同的操作，重复的次数也有所不同。其次，可能需要多个浮点运算单元，如果要发射的指令有数据冒险或者使用运算单元的结构冒险，就应该进行停顿。

在这一节中，假设我们的 MIPS 有 4 个独立的运算单元：

1. 主定点操作单元，负责 load、store、定点 ALU 操作和转移操作。
2. 浮点和定点乘法单元。
3. 浮点加法单元，处理浮点加、减法及定点数和浮点数之间的转换。
4. 浮点和定点除法单元。

图 A.29 给出了上述操作的流水线结构，并假设运算单元在执行阶段（EX）没有采用流水线技术。因为 EX 段没有采用流水线，所以直到前一条指令的 EX 段结束之前，其他使用该运算单元的指令都不能执行。另外，如果某条指令不能执行到 EX 段，在这条指令之后的整个流水线都会被停顿。

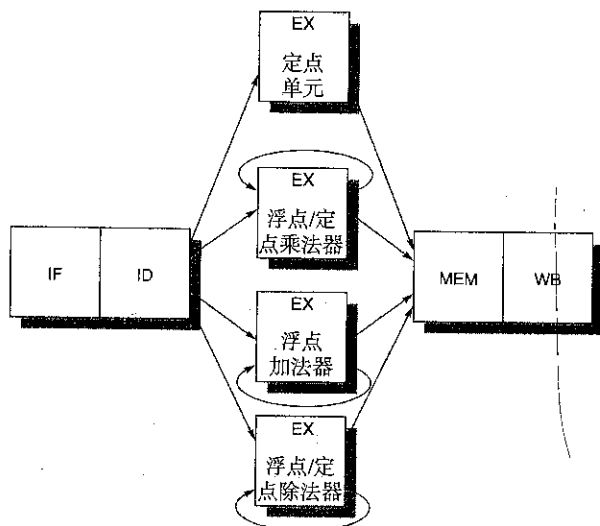


图 A.29 增加了3个非流水浮点功能单元的MIPS流水线。因为每个时钟周期只发射一条指令，所以所有的指令都经过定点操作的标准流水线。浮点运算要在EX段进行循环。在浮点操作结束EX段之后，才能进入MEM和WB段以完成指令

事实上，中间的执行过程并不需要像图 A.29 那样在 EX 单元进行循环，而是 EX 单元的流水段有超过一个时钟周期的延迟。我们可以将图 A.29 所示的浮点流水线结构进行扩展，允许某些段流水线化，并且多条指令同时操作。为了描述这样的流水线，我们必须定义运算单元的延迟和启动间距或重复间距。和以前一样，我们把延迟定义为介于一条产生结果的指令和使用该结果的指令之间的时钟周期数，启动（或重复）间距定义为执行两个给定类型的操作之间必须经历的时钟周期数。例如，我们使用如图 A.30 所列出的延迟和启动间距。

功能单元	延迟	启动间距
定点 ALU	0	1
数据存储器（定点和浮点 load）	1	1
浮点加	3	1
浮点乘（包括定点乘）	6	1
浮点除（包括定点除）	24	25

图 A.30 运算单元的延迟和启动间距

按照延迟的上述定义，定点 ALU 操作的运算结果在下一个时钟周期就可以使用。所以它的延迟为 0；load 的结果隔一个周期就可以使用，所以 load 的延迟为 1。因为大多数操作在 EX 段开始时就开始使用它们的操作数。所以，延迟通常是指令在产生结果的 EX 段之后还要再执行的段数。例如，ALU 运算是 0 步，load 是 1 步。一个主要的例外是 store，它在一个周期后就可以使用存储的值，store 的延迟以这个被 store 存储的值为依据，而不是以基地址寄存器为依据。所以其延迟要少一个周期。流水线延迟基本上等于执行流水线的深度减去一个时钟周期，而执行流水线的深度为从

EX 的开始到产生结果之间的段数。这样，以上面的流水线为例，浮点加的段数为4，而浮点乘的段数为7。为了提高时钟频率，设计者们要在流水线的每一个段设置较少的逻辑层。这就使复杂的操作所需的段数增多了。可以看出，提高时钟频率的代价是增加了操作的流水线时延。

在图 A.30 所示的流水线结构中，允许最多4个浮点加、7个浮点/定点乘和一个浮点除操作同时执行。图 A.31 说明如何通过扩展图 A.29 得到流水线。在图 A.31 中，重复间距的实现方法是：增加一些额外的流水段，而且这些段可以被额外的流水线寄存器分离出来。因为各单元彼此独立，所以我们将各段分别命名。需要多个时钟周期的流水段，如除法单元的段，被进一步细分，以指明这些段的时延。但是，因为它们不是完整的段，只能有一个操作是活动的，也可以使用类似本章中前面使用过的示意图来表示流水线结构。图 A.32 所示为一组独立的浮点运算，浮点 load 和浮点 store 操作。自然浮点操作的长时延增加了 RAW 冒险和该冒险造成的停顿，这些我们将在本节的稍后部分涉及到。

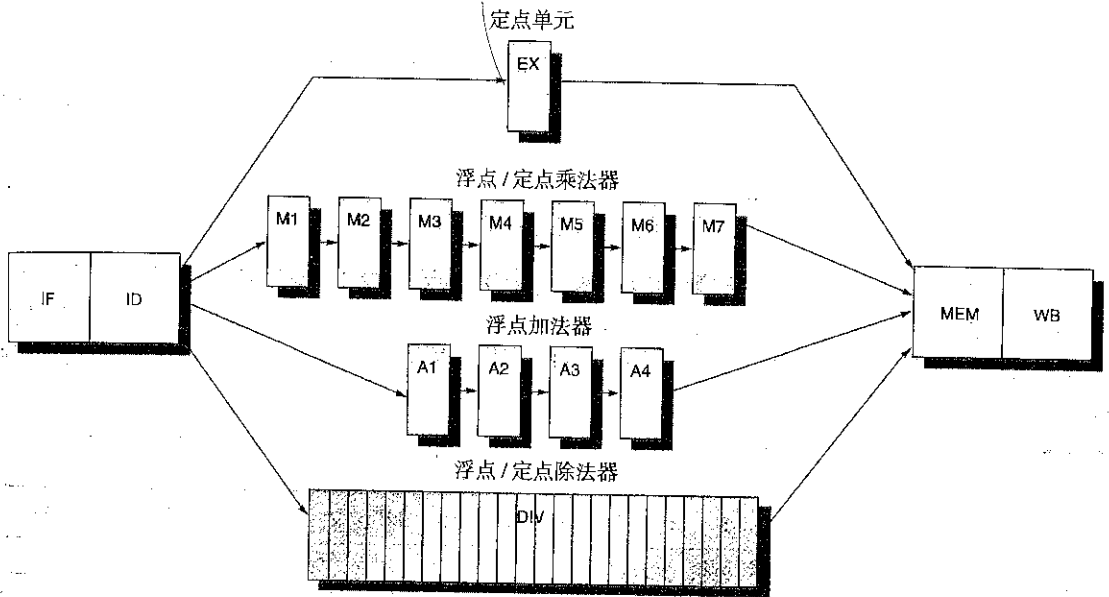


图 A.31 一条支持多个浮点操作同时执行的流水线。浮点乘法单元和浮点加法单元是全流水化的，深度分别为7段和4段。浮点除法单元没有采用流水线，它需要15个时钟周期才能完成。对于一条指令，从浮点操作的发射到不产生RAW冒险的情况下使用该操作的结果之间的时延，取决于在执行段中所花费的周期数。例如，浮点加之后的第4条指令就可以使用该浮点加的结果。对于定点 ALU 操作，执行流水线的深度总是1，下一条指令就可以使用它的运算结果。浮点 load 操作和定点 load 操作都在 MEM 段完成，这就意味着存储器必须在一个时钟周期内提供 32 位或者 64 位的数据

要理解图 A.31 中的流水线结构，需要介绍以下附加的流水线寄存器（如 A1/A2, A2/A3, A3/A4）和与这些寄存器的连接关系。ID/EX 寄存器必须扩展到能把 ID 连接到 EX, DIV, M1 和 A1，我们可以将寄存器中与下一个段中的某一个相关联的端口用符号 ID/EX, ID/DIV, ID/M1 和 ID/A1 来标识。ID 和所有段间的流水线寄存器可以被认为是逻辑上独立的寄存器，并且事实上可以用独立的寄存器来实现。由于在同一时刻，一个流水段内只能有一个操作，那么控制信息就可以在段的开头部分与寄存器联系起来。

MUL.D	IF	ID	<i>M1</i>	M2	M3	M4	M5	M6	M7	MEM	WB
ADD.D		IF	ID	<i>A1</i>	A2	A3	A4	MEM	WB		
L.D			IF	ID	<i>EX</i>	MEM	WB				
S.D				IF	ID	<i>EX</i>	<i>MEM</i>	<i>WB</i>			

图 A.32 一组独立的浮点操作的流水线时序。斜体印刷的段需要数据。黑体印刷的段已经得到结果。由于浮点 load 和 store 需要 64 位宽度的存储器数据通路, 所以流水线时序中只是定点 load 或 store

长时延流水线中的冒险和直通

对于图 A.31 中所示的那种流水线, 冒险检测和设置直通有一些新的特点。

1. 由于浮点单元没有采用流水线结构, 因此可能出现结构冒险。当检测到这种冒险时, 正在执行中的指令可能被停顿。
2. 因为各种指令的执行时间不同, 所以在一个周期内所需要的写寄存器的次数可能大于 1。
3. 因为指令不再按顺序到达 WB, 所以可能引起 WAW 冒险。另外, 由于寄存器读操作总在 ID 段进行, 因此不存在 WAR 冒险。
4. 指令可以按照与它们发射时不同的顺序执行完成, 这将引起一些意外的问题。
5. 由于操作的长时延, 冒险所引起的停顿将更加频繁。

操作的长时延所引起的停顿次数的增加基本上与定点流水线相同。在描述浮点流水线所产生的新问题并给出解决方案之前, 让我们先看看 RAW 冒险的潜在影响。图 A.33 所示为一个典型的浮点操作代码序列及其造成的停顿, 在这一节的末尾, 我们将在 SPEC 子集上测试这个浮点流水线的性能。

指令	时钟周期																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
L.D F4,0(R2)	IF	ID	EX	MEM	WB												
MUL.D F0,F4,F6		IF	ID	停顿	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D F2,F0,F8			IF	停顿	ID	停顿	停顿	停顿	停顿	停顿	停顿	A1	A2	A3	A4	MEM	WB
S.D F2,0(R2)					IF	停顿	停顿	停顿	停顿	停顿	停顿	ID	EX	停顿	停顿	停顿	MEM

图 A.33 由 RAW 冒险引起停顿的一段典型浮点操作代码。相对于深度较浅的定点流水线, 较长的浮点流水线使得停顿出现的频率增加。在假设流水线是设置有旁路的前提下, 这一段代码中的每一条指令都与前一条相关, 并且在数据可用时, 马上就继续执行。S.D 操作必须额外停顿一个周期, 以使它的 MEM 和 ADD.D 操作不冒险, 这可以用附加硬件很容易地实现

现在来看上面的(2)和(3)两条所描述的由写操作引起的问题。如果我们假设浮点寄存器堆只有一个写端口, 那么浮点操作序列可能引起寄存器写端口的冒险, 正如在浮点操作的同时进行浮点 load 时一样。例如, 考虑图 A.34 所示的流水线序列, 在周期 11, 全部 3 条指令都到达 WB, 并且都要求写寄存器堆。在只有一个寄存器写端口的情况下, 机器必须使指令的执行串行化。单个寄存器端口意味着有结构冒险。我们可以通过增加写端口数来解决这个问题, 但是由于额外的写端口很少被用到, 这种方案就缺乏吸引力, 这是因为绝大多数情况下只需要一个写端口。因此, 我们选择的方案是检测对写端口的访问, 并把多个同时的写操作看做结构冒险。

指令	时钟周期										
	1	2	3	4	5	6	7	8	9	10	11
MUL.D F0,F4,F6	IF	ID	MI	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
ADD.D F2,F4,F6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
L.D F2,0(R2)							IF	ID	EX	MEM	WB

图 A.34 在周期11, 3条指令试图同时完成写浮点寄存器堆的操作。但是这并不是最糟糕的情况, 由于更早一些启动的浮点单元中的除法操作也可能在这个周期内完成, 因此冒险将更难解决

有两种办法可以解决这种冒险。第一种办法是在ID段跟踪写端口的使用, 并且像对待其他结构冒险一样在指令发射前就停顿它。跟踪写端口的使用是通过一个移位寄存器来实现的, 它指明已经发射的那些指令将在何时使用寄存器堆。如果在ID中的指令需要与已经发射的指令同时使用寄存器堆, 在ID中的指令就被停顿一个周期发射。在每个时钟周期内, 保留(reservation)寄存器都移位一次。这种实现方法的优点是, 它维持了只在ID段停顿指令的特性, 代价是增加了一个移位寄存器和写冲突的判断逻辑。这一节中, 我们都假定使用这种方案。

另一种办法是在有冒险的指令试图进入MEM和WB段时再停顿它。如果我们等产生冒险的指令达到MEM段前再停顿它们, 就可以有选择地停顿指令。一个简单但有时并不是最优的启发式算法是给那些时延最长的单元以优先权, 因为它们最容易导致其他指令被停顿, 从而产生RAW冒险。这种方案的优点是直到进入容易检测冒险的MEM或WB段以前, 我们都不需要检测冒险。缺点是使流水线的控制变得复杂, 因为可能在两个不同的地方产生停顿。要注意这一进入MEM前的停顿将引起EX, A4或M7等段被填满, 这就可能会迫使停顿在流水线中的操作被反向传送。同样, WB段之前的停顿可能会使MEM反向传送。

另外的一个问题是可能会引起WAW冒险。考虑图A.34中的例子, 我们看到这种情况是存在的。如果LD指令提前一个周期发射, 且目标操作数是F2, 那么它将引起WAW冒险, 因为它比ADD.D指令早一个周期写F2。注意到只有ADD.D指令的结果在没有任何指令使用它的情况下就被覆盖时, 这种冒险才会出现。如果在ADD.D和LD之间有对F2的引用, 流水线就可能因为RAW冒险而被停顿, 使得LD要等到ADD.D执行完毕后才能发射。对我们的流水线来说, WAW冒险只有在执行一条无用指令时才会出现。但是, 我们仍然必须检测到它, 并且操作序列结束时, LD的结果写到F2中(就像我们在A.8节中看到的, 这样的序列确实可能出现)。

有两种可能的方案可以处理WAW冒险。第一种方案是延迟load指令, 直到ADD.D指令进入MEM段。第二种方案是通过检测冒险并改变控制使ADD.D指令不写其结果。然后LD指令就能够立刻发射了。因为这种冒险极少出现, 所以两种方案都很有效。你可以挑选容易些的方案来实现。在两者中, 都可以在发射LD指令的ID段检测到冒险, 然后停顿LD指令或使ADD.D成为一个空操作。困难的情形是LD指令是否要在ADD.D指令之前结束。因为这需要流水线的长度和ADD.D指令现在的位置。幸运的是, 这个代码序列(两个写操作之间没有读操作)比较特殊。所以, 我们采用一个简单的解决方案。如果一条在ID段中的指令试图与已经发射的指令同时写一个寄存器, 就

不将其发射到 EX 段。在 A.7 节中，我们将看到附加的硬件如何消除这种冒险引起的停顿。下面，我们将浮点流水线中的冒险和发射逻辑的实现细节结合到一起。

在检测可能的冒险时，我们既要考虑浮点指令的冒险，又要考虑浮点指令和定点指令的冒险。除了浮点 load-store 指令和浮点 - 定点寄存器传送指令之外，浮点寄存器和定点寄存器是相互独立的。所有的定点指令都在定点寄存器内进行操作，而所有的浮点操作只在它们自己的寄存器内进行。这样，在检测浮点指令和定点指令的冒险时，我们只需要考虑浮点 load-store 指令及浮点寄存器传送指令与定点指令之间的冒险。这种流水线控制的简化是定点数据和浮点数据具有不同的寄存器堆的方法所附带的优点。主要的优点是使寄存器数目加倍，而每个寄存器堆并没有增大，并且在不增加寄存器端口的情况下，增加了带宽。主要的缺点除了需要一个额外的寄存器堆外，还有偶尔出现在两个寄存器之间的数据传送带来的较小开销。假定流水线在 ID 段检测所有的冒险，那么，在一条指令发射前，必须完成如下三种检查：

1. **检查结构冒险：**所需要的运算单元都不忙（在该流水线中，只有除法需要），并且确保寄存器写端口在需要时可用。
2. **检查 RAW 数据冒险：**源寄存器与即将使用的流水线寄存器中目标寄存器（当指令需要结果时，这样的寄存器是不可用的）不相同。在此必须做许多检查。这取决于源指令和目标指令，源指令决定了结果何时可用，目标指令决定了何时需要它的值。例如，ID 段中的指令是一个源寄存器为 F2 的浮点操作，那么 F2 就不能作为 ID/A1，A1/A2，A2/A3 的目标寄存器。这正符合当 ID 中的指令需要结果时，浮点加指令还不能结束的情况（ID/A1 是 ID 到 A1 的输出指令寄存器的接口）。如果我们试图允许除法的最后几个周期重叠，那么对除法的处理就有些麻烦。因为我们必须在除法将要结束时，专门处理这样的情况。实际上，出于简化发射测试的目的，设计者会忽略这种优化。
3. **检查 WAW 冒险：**判断是否有 A1，…，A4，D，M1，…，M7 中的指令与这条指令有相同的目标寄存器。如果有，就在 ID 段中停顿这条指令的发射。

尽管由于多周期的浮点操作使冒险检测变得复杂了，但是有关概念与 MIPS 定点流水线是相同的。直通逻辑也是如此，直通可以通过检查在 EX/MEM，A4/MEM，M7/MEM，D/MEM 或 MEM/WB 寄存器中的目标寄存器是否是一条浮点指令的源寄存器来实现。如果是，那么相应的数据选择器就开始工作，以使它选择发送的数据。在实际应用中，你可以像检测 RAW 冒险和 WAW 冒险那样自己动手设计旁路检测逻辑。

多周期浮点操作也给我们的异常处理机制带来了一些问题。我们将在下面一节中进行讨论。

维护精确异常处理

由运行时间长的指令引起的另外一个问题可以由下面的这段代码序列来说明：

```
DIV.D    F0,F2,F4
ADD.D    F10,F10,F8
SUB.D    F12,F12,F14
```

这段代码看起来很简单，各条指令间没有数据相关关系，但是由于先发射的指令可能在后发射的指令之后执行完毕，问题就产生了。在这个例子中，我们可以预料到，ADD.D 和 SUB.D 指令可以在 DIV.D 指令之前执行完毕。这种情况称为乱序完成（out-of-order completion）。当流水线中有运行时间长的指令存在时，乱序完成的情况就会经常发生（见 A.7 节）。可是，既然冒险检测防止了任何有数据相关的指令间的相互干扰，为什么乱序完成还会成为一个问题呢？假设 SUB.D 指令和

ADD.D 指令已经完成,但是在 DIV.D 指令结束之前的一段时间内产生了一个浮点算术异常,其结果将是不精确的异常,这正是我们试图要避免的。这个问题看起来可以像对待定点流水线那样来处理,通过清空浮点流水线来解决,但是,异常可能在一个不能采用这种方法的位置上出现。例如,如果 DIV.D 指令在加法操作结束之后产生了一个浮点算术异常,我们即使在硬件级别上也不能获得精确的异常。事实上,由于 ADD.D 指令破坏了一个源操作数,我们即使在软件的帮助下,也不能恢复到执行 DIV.D 指令之前的状态。

这个问题的产生是由于指令完成的顺序与它们发射时的顺序不同。有 4 种可能的方案解决乱序完成。一种方案是忽略这个问题,满足于不精确的异常。这种方案在 20 世纪 60 年代和 20 世纪 70 年代早期被使用,现在在某些巨型机上仍在使用。在这些机器中,某些类型的异常被禁止,或在硬件级别上处理,而不是中停顿流水线。对于大多数现在制造的处理器来说,很难再采用这种策略,因为虚拟存储器和 IEEE 浮点标准等要求通过软硬件相结合的方法实现精确异常。正如我们在前面提到的,近期制造的处理器通过引入两种异常模式解决了这个问题:一个是快速但精确的模式,一个是慢速精确模式,慢速精确模式或者通过一个模式开关来实现,或者通过插入一条显示指令测试浮点异常来实现。在这两种情况下,浮点流水线中所允许的重叠和重排序的数目要严格限制,以使同一时刻只有一条浮点指令是活动的。这种方案被用于 DEC Alpha 21064 和 21164, IBM Power 1 和 Power 2 以及 MIPS R8000 处理器中。

第二种方案是采用缓冲存储器缓存操作结果,直到所有在该操作之前发射的操作都执行完毕。有些 CPU 的确使用这种方案,但是当不同操作间所需的时间相差很大时,由于需要缓存的结果很多,这种方案就变得代价过高了。而且,在等待较长指令时,队列中的结果必须被旁路,以便继续发射后面的指令,这需要大量的比较器和一个非常大的数据选择器。

在这种基本方案的基础上,有两个不同的变种。第一种是在 CYBER 180/990 中使用的历史文件。在历史文件中记录寄存器的原始值。当异常出现且状态必须返回到一些乱序完成指令之前时,寄存器的原始值就可以从历史表中读出并得到恢复。类似的技术在 VAX 之类的机器中用于自动增量和自动减量寻址。另一种方案是由 J. Smith 和 A. Pleszkun 在 1988 年提出来的未来文件方案,这种方案保存寄存器的新值,当所有该指令之前的指令都执行完毕之后,主寄存器堆根据未来文件更新。对一个特定的异常,主寄存器堆中保存了被中断状态的精确信息。在第 2 章中,这种思想得到了推广,用于在保持精确异常的同时允许重叠和重排序的机器中,如 PowerPC 620 和 MIPS R10000 处理器中。

第三种方案是允许异常不十分精确,但是要保存足够多的信息以使陷阱处理程序为这个异常建立一个精确的序列。这意味着要知道流水线中有哪些操作以及它们的 PC 值,然后在处理完异常之后,由软件完成那些在刚完成的指令之前的指令,随后重启指令序列。考虑如下最坏情况的代码序列:

指令₁——一条长时间运行但最后发生中断的指令。

指令₂, ..., 指令_{n-1}——一系列未完成的指令。

指令_n——一条执行完成的指令。

在给出了流水线中的各指令的 PC 和异常返回的 PC 之后,软件就能确定指令₁和指令_n的状态了。因为指令_n已经完成,我们希望从指令_{n+1}开始重启执行。在处理完异常之后,软件必须模拟执行指令₁到指令_{n-1}。然后就可以从异常处理中返回,再从指令_{n+1}重启。处理程序正确执行这些指令的复杂性是这种方案的主要困难。

简单的MIPS流水线有一个重要的简化: 如果指令₂到指令_n都是定点指令, 那么如果指令₁已经执行完了, 则指令₂到指令_{n-1}也一定执行完了。这样, 只有浮点操作需要被处理。为使这个方案易于管理, 就要对执行时所允许重叠的浮点指令数加以限制。例如, 如果我们只重叠两条指令, 那么只有发生中断的指令需要由软件完成执行。如果浮点流水线很深或者有很多个浮点运算单元, 这种限制可能会降低吞吐率。这种方案被用于SPARC结构中, 以允许浮点操作和定点操作重叠。

最后一种方案是一种混合的方案。只有在确信正要发射的指令之前的那些指令都将不发生异常地执行完毕时, 才允许该指令继续发射。这就确保了当异常出现时, 在被中断的指令之后的那些指令都不会完成, 而在它之前的所有指令都已经完成。这意味着有时要靠停顿CPU来维持精确的异常。为了实现这种方案, 浮点运算单元必须在EX段的前期(在MIPS流水线的前三个周期中)确定是否会出现异常, 以防止后面的指令继续执行。这种方案被用于MIPS R2000/R3000, R4000和Intel Pentium处理器中, 在附录I中有对这种方案的深入讨论。

MIPS浮点流水线的性能

图A.31所示的MIPS浮点流水线可能产生除法单元的结构冒险产生的停顿和由RAW冒险引起的停顿, 它还可能引起WAW冒险的停顿, 但实际上极少出现。图A.35中列出了基于具体测试标准的每种浮点操作的停顿周期数(例如, 每个浮点基准测试程序的第一条表示浮点结果被停顿的个数, 包括浮点加、减或比较)。我们可以看到各种浮点操作的停顿周期数是其操作单元延迟周期数的46%~59%。

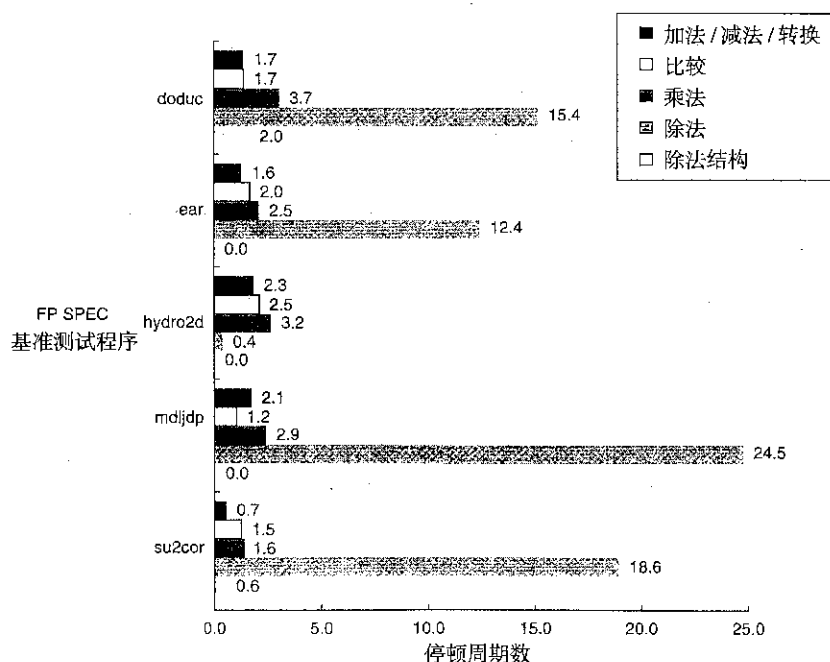
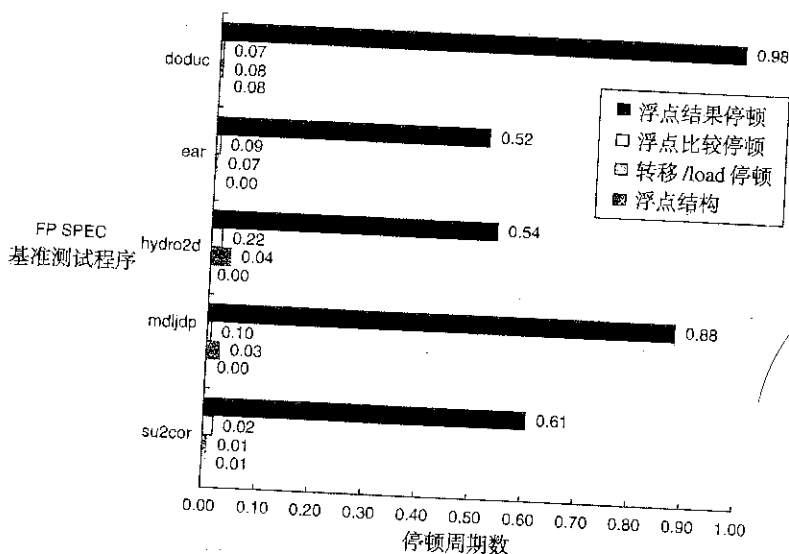


图 A.35 每种主要浮点操作的停顿周期数 (SPEC基准测试程序)。除了除法单元的结构冒险之外, 这些数据不依赖于操作出现的频率, 而只依赖于它的时延和结果被使用之前的时钟周期数。RAW 冒险引起的停顿数大致反映了浮点单元的时延。例如, 每个浮点加、减或转换的平均停顿数为 1.7 个周期, 或者说是时延周期数 (3 个周期) 的 56%。同样, 乘法和除法的平均停顿数是 2.8 和 14.2, 是各自时延周期数的 46% 和 59%。由于除法出现的频率很低, 所以除法的结构冒险很少见。

图A.36给出了在5种浮点SPECfp基准测试程序下，定点与浮点的停顿差别。有4种停顿：浮点结果停顿、浮点比较停顿、load与转移延迟和浮点结构延迟。编译器尽量在调度转移延迟之前调度load与浮点延迟。每条指令的总停顿数目分布在0.65~1.21之间。



图A.36 在5种SPEC89浮点基准测试程序下，DLX浮点流水线中出现的停顿。每条指令的总停顿数从su2cor程序的0.65到doduc程序的1.21，平均为0.87。浮点结果停顿在所有情况下都占统治地位，平均每条指令有0.71个停顿，或者说是停顿周期总数的82%。比较操作的每条指令平均产生0.1的停顿，是第二大来源。对于doduc程序，除法的结构冒险是唯一的因素。

A.6 综合：MIPS R4000 流水线

本节我们将考察MIPS R4000系列处理器（包含4400）的流水线结构及其性能。R4000实现MIPS64指令系统，但是使用的流水线比我们的5段定点流水线和浮点流水线都要深。这条更深的流水线通过把5个段的定点流水线拆分成8个段，因此获得了更高的时钟频率。因为Cache访问对时间的要求很严，所以，多出来的段主要是把Cache访问分成了几个流水段。这种深度的流水线有时称为超流水线。

图A.37是简化了的8段流水线结构。图A.38表示流水线中相邻指令重叠执行的过程。请注意，尽管访问存储器需要几个时钟周期，但是存储器是完全流水的，所以仍可以每个周期启动一条指令。实际上，流水线中使用的数据在Cache命中检测之前就已经获得了；第5章中将详细解释这种方案是怎样实现的。

每个段的功能如下：

- IF：取指令的前半部，此时选择PC，并开始访问指令Cache。
- IS：取指令的后半部，完成对指令Cache的访问。
- RF：指令译码，并读寄存器堆；冒险检查并对指令Cache是否命中进行检查。
- EX：执行，包括有效地址计算、ALU操作、转移目标地址计算和判断转移条件。
- DF：读数据，访问数据Cache的前半部。
- DS：读数据的后半部，完成对数据Cache的访问。

- TC: 检查标志, 确定数据 Cache 是否命中。
- WB: 为 load 和寄存器-寄存器操作返回结果。

除了增大对直传硬件的需求, 这种长时延流水线还增加了 load 和转移延迟。如图 A.38 所示, load 的延迟为 2 个周期, 因为在 DS 的末尾能得到数据。图 A.39 是紧跟 load 的指令使用数据时, 流水线调度的简化图。该图说明在 3 或 4 个周期之后要求使用 load 的结果时, 就需要进行直通。

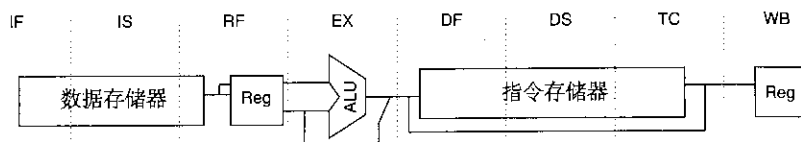


图 A.37 R4000 的 8 段流水线结构使用流水化的指令与数据 Cache。流水段的功能在图中已经说明。虚线既是流水段的边界, 又是流水线寄存器的位置。指令实际在 IS 结束时得到, 但是在 RF 段读寄存器时才检测指令的标志。因此, 在图中把指令存储器一直画在 RF 段中。访问数据存储器时需要到达 TC 段, 因为直到确定 Cache 是否命中后才能把数据写入寄存器

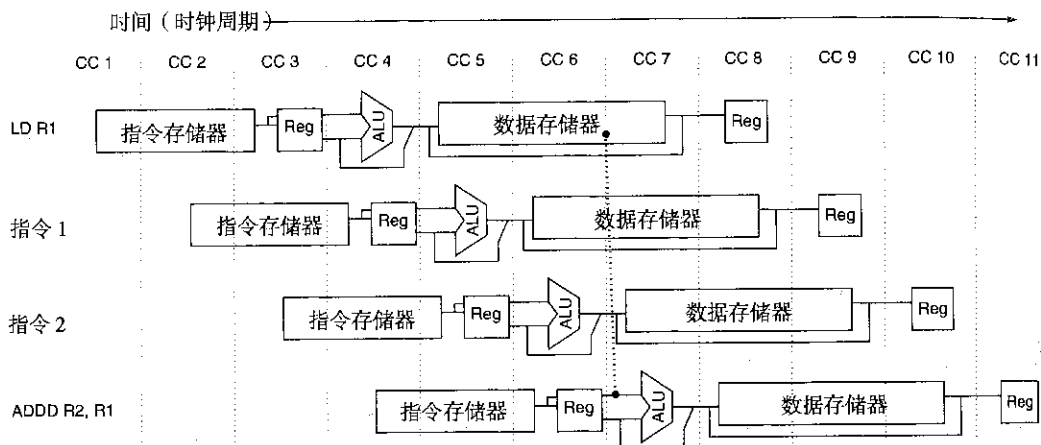


图 A.38 R4000 定点流水线的结构产生 2 个周期长的 load 延迟。2 个周期长的延迟能满足要求, 因为数据在 DS 段的末尾就可以使用, 而且可以旁路。如果在 TC 段发现了 Cache 不命中, 流水线就要后退一个周期, 以得到正确数据

图 A.40 表明基本的转移延迟是 3 个周期, 因为转移条件是在 EX 段计算出的。MIPS 系统结构的转移延迟却只有一个周期, 因为 R4000 对转移延迟的剩余两个周期采用了预测未选中策略。如图 A.41 所示, 未选中转移的延迟仅为一个周期, 而被选中转移是一个单周期延迟槽再跟两个空周期。前面说过, 指令集提供了一条类似于转移的指令, 它帮助填充转移延迟槽。流水线锁定器在转移被选中时强制加入了两个周期的转移停顿, 并在使用 load 结果时加入数据冒险停顿。

深度流水线除了要增加 load 与转移的延迟之外, 还增加了 ALU 操作的直传级数。在我们的 5 段 MIPS 流水线中, 两条寄存器-寄存器 ALU 指令之间的直传可以发生于 ALU/MEM 或 MEM/WB 寄存器。在 R4000 流水线中, ALU 旁路有 4 个可能的来源: EX/DF, DF/DS, DS/TC 和 TC/WB。

指令	时钟								
	1	2	3	4	5	6	7	8	9
LD R1,...	IF	IS	RF	EX	DF	DS	TC	WB	
DADD R2,R1,...		IF	IS	RF	停顿	停顿	EX	DF	DS
DSUB R3,R1,...			IF	IS	停顿	停顿	RF	EX	DF
OR R4,R1,...				IF	停顿	停顿	IS	RF	EX

图 A.39 紧随load使用数据时,产生2个周期的停顿。正常的直传可以用在2个周期后,所以DADD指令和DSUB指令能在停顿后通过直传得到所需要的数据。而OR指令是从寄存器堆得到数据的。因为load后的两条指令可以与load不相关,也就没有停顿,所以旁路可以传到load后的3个甚至4个周期

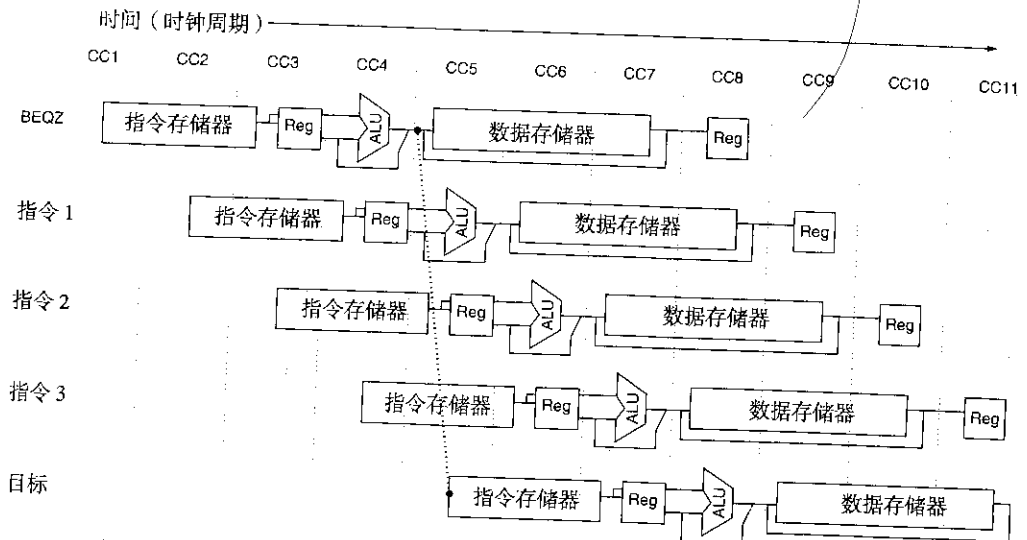


图 A.40 基本的转移延迟是3个周期,因为转移条件是在EX段形成的

指令	时钟周期								
	1	2	3	4	5	6	7	8	9
转移指令	IF	IS	RF	EX	DF	DS	TC	WB	
延迟槽		IF	IS	RF	EX	DF	DS	TC	WB
停顿			停顿	停顿	停顿	停顿	停顿	停顿	停顿
转移目标				IF	IS	RF	EX	DF	

指令	时钟周期								
	1	2	3	4	5	6	7	8	9
转移指令	IF	IS	RF	EX	DF	DS	TC	WB	
延迟槽		IF	IS	RF	EX	DF	DS	TC	WB
转移指令 + 2			IF	IS	RF	EX	DF	DS	TC
转移指令 + 3				IF	IS	RF	EX	DF	DS

图 A.41 图的上半部分表示被选中转移有一个周期的延迟槽和两个周期的停顿,而下半部分表示未选中转移只有一个延迟槽。这里的转移指令可以是普通的转移,也可以是隐含转移,当转移未选中时,要把延迟槽中的指令执行结果取消

浮点流水线

R4000的浮点单元由3个功能单元组成：浮点除法器、浮点乘法器和浮点加法器。在乘法、除法的最后一步需要只用加法器。双精度浮点运算的运行时间变化很大，可以少至2个周期（求反），多至112个周期（求平方根）。此外，不同单元的启动速度也不一样。如图A.42所示，可以把浮点运算单元分成8个段，这些段以不同的顺序组合来执行不同的浮点运算模式。

段	功能单元	描述
A	浮点加法器	尾数加流水线
D	浮点除法器	除法流水段
E	浮点乘法器	异常测试流水段
M	浮点乘法器	乘法第一个流水段
N	浮点乘法器	乘法第二个流水段
R	浮点加法器	舍入流水段
S	浮点加法器	操作数移位流水段
U		拆分浮点数

图 A.42 R4000 浮点流水线的 8 个流水段

图中每个段都有一个副本，而不同的指令对一个流水段可能不执行也可能按不同顺序执行多次。图 A.43 列出了最常见的双精度浮点操作的延迟、启动间隔和流水段。

浮点指令	延迟	启动间隔	流水段
加减法	4	3	U, S + A, A + R, R + S
乘法	8	4	U, E + M, M, M, M, N, N + A, R
除法	36	35	U, A, R, D ²⁷ , D + A, D + R, D + A, D + R, A, R
求平方根	112	111	U, E, (R + A) ¹⁰⁸ , A, R
求补	2	1	U, S
求绝对值	2	1	U, S
浮点数比较	3	2	U, A, R

图 A.43 浮点操作的延迟和启动间隔都依赖于具体操作必须使用的浮点单元的流水段。图中的延迟假设目标指令为浮点操作；当目标指令是 store 时，延迟要少一个周期。流水段表示各种段的工作顺序。图中符号 S + A 表示在一个时钟周期内 S 段和 A 段都被使用。符号 D²⁸ 表示 D 段在这一行里使用了 28 次

从图 A.43 中可以确定一系列不相关的浮点操作是否可以无停顿地发射。如果指令序列表明一个公用的流水段有冲突发生，则需要一个停顿。图 A.44、图 A.45、图 A.46 和图 A.47 分别列出了 4 种常见的双指令序列：先乘法后加法、先加法后乘法、先除法后加法和先加法后除法。图中画出了第二条指令所有可能的启动位置，以及它如果从那里启动会被发射还是停顿。当然，也可能有 3 条活跃的指令，此时停顿会更长，图也会更复杂。

R4000 流水线的性能

我们现在考察 R4000 流水线结构在 SPEC92 基准测试程序下的停顿情况。流水线停顿主要有 4 个原因：

1. **load 停顿**：由 load 操作产生的在 load 之后一或两个周期的停顿。

2. 转移停顿：转移延迟槽中未填入指令或填入了被取消的指令，或每个被选中的转移带有的两个周期的停顿。
3. 浮点结果停顿：浮点操作数的 RAW 冒险形成的停顿。
4. 浮点结构停顿：浮点流水线中，因功能单元冲突而引起的发射延迟。

图 A.48 和图 A.49 分别用柱形图和表格列出了 R4000 流水线在 10 个 SPEC92 基准测试程序下的分类 CPI。

操作	发射/停顿	时钟周期											
		0	1	2	3	4	5	6	7	8	9	10	11 12
乘法	发射	U	E + M	M	M	M	N	N + A	R				
加法	发射		U	S + A	A + R	R + S							
	发射			U	S + A	A + R	R + S						
	发射				U	S + A	A + R	R + S					
	停顿					U	S + A	A + R	R + S				
	停顿						U	S + A	A + R	R + S			
	发射							U	S + A	A + R	R + S		
	发射								U	S + A	A + R	R + S	

图 A.44 在周期0发射的浮点乘法后面跟着一条从周期1到7的浮点加法指令。第二列表示后面第 n 周期的指令是发射还是停顿。引起停顿的段用黑体表示。请注意该表只能用于乘法与它后面第1到7周期的加法之间的作用关系。在本例中，在乘法后4个或5个周期启动的加法操作将被停顿，其余的都可以发射。请注意，想在第5个周期启动的加法停顿一个周期后就可以启动；而想在第4个周期启动的加法，在停顿一个周期后到达第5个周期时，仍需要再停顿一个周期，所以这条指令被停顿两个周期

操作	发射/停顿	时钟周期											
		0	1	2	3	4	5	6	7	8	9	10	11 12
乘法	发射		U	S + A	A + R	R + S							
加法	发射			U	E + M	M	M	N	N + A	R			
	发射				U	M	M	M	N	N + A	R		

图 A.45 乘法在加法之后的序列通常不会停顿，因为执行时间很短的加法指令在乘法指令到达一个流水段之前，就已经结束了在这个段的执行过程

从图 A.48 和图 A.49 的数据中可以看到深度流水线的代价。R4000 的转移延迟远比经典 5 段流水线的长，因此用于转移的时钟周期数增多了，尤其是转移频率较大的定点程序。浮点程序的一个有趣结果是浮点功能单元的延迟比结构冒险产生的停顿更多，这是由于启动间距的限制和不同浮点指令使用相同浮点单元时的冲突这两个原因造成的。于是，首要的任务不是实现更长的流水或使用更多的功能单元，而是减少浮点操作的延迟。当然，由于很多潜在的结构冒险隐藏在数据冒险之后，所以减少延迟可能会增加结构冒险。

操作	发射/停顿	时钟周期											
		25	26	27	28	29	30	31	32	33	34	35	36
除法	在周期0 发射...	D	D	D	D	D	D+A	D+R	D+A	D+R	A	R	
加法	发射		U	S+A	A+R	R+S							
	发射			U	S+A	A+R	R+S						
	停顿				U	S+A	A+R	R+S					
	停顿					U	S+A	A+R	R+S				
	停顿						U	S+A	A+R	R+S			
	停顿							U	S+A	A+R	R+S		
	停顿								U	S+A	A+R	R+S	
	停顿									U	S+A	A+R	R+S
	发射										U	S+A	A+R
	发射											U	S+A
	发射												U

图 A.46 浮点除法可能使一条在该除法指令结束之前不久的加法指令停顿。除法指令从周期0开始,在周期35结束,图中仅画出了除法操作的后面10个周期。当除法和加法反复使用舍入部件时,从周期28至周期33,加法操作被停顿。注意到加法实际上从周期28开始到周期36都被停顿。如果加法操作在除法操作之后立即就开始,它将没有冲突,因为加法操作在除法操作所需要的公共段开始之前就已经完成了。像前面的图那样,在这个例子中,假设加法操作的U段恰好在周期26与周期35之间开始

		时钟周期												
操作	发射/停顿	0	1	2	3	4	5	6	7	8	9	10	11	12
加法	发射	U	S + A	A + R	R + S									
除法	停顿		U	A	R	D	D	D	D	D	D	D	D	D
	发射			U	A	R	D	D	D	D	D	D	D	D
	发射				U	A	R	D	D	D	D	D	D	D

图 A.47 双精度加法后面有双精度除法。如果除法在加法后一个周期启动,除法就被停顿,否则没有冲突

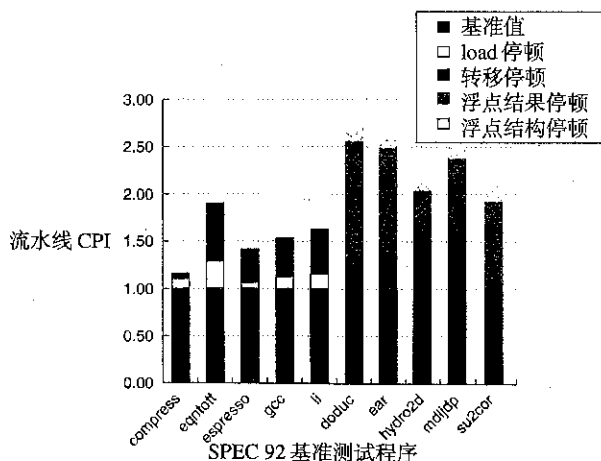


图 A.48 假设使用理想 Cache,在 SPEC92 的 10 个基准测试程序下的流水线 CPI。流水线 CPI 的变化范围为 1.2~2.8。最左边的 5 个程序是定点程序,这些程序中转移延迟是影响 CPI 的主要因素。最右边的 5 个程序是浮点程序,浮点结果停顿是影响 CPI 的主要因素

基准程序	流水线 CPI	load 停顿	转移停顿	浮点结果停顿	浮点结构停顿
compress	1.20	0.14	0.06	0.00	0.00
eqntott	1.88	0.27	0.61	0.00	0.00
espresso	1.42	0.07	0.35	0.00	0.00
gcc	1.56	0.13	0.43	0.00	0.00
li	1.64	0.18	0.46	0.00	0.00
定点平均	1.54	0.16	0.38	0.00	0.00
doduc	2.84	0.01	0.22	1.39	0.22
mdljdp2	2.66	0.01	0.31	1.20	0.15
ear	2.17	0.00	0.46	0.59	0.12
hydro2d	2.53	0.00	0.62	0.75	0.17
su2cor	2.18	0.02	0.07	0.84	0.26
浮点平均	2.48	0.01	0.33	0.95	0.18
总平均	2.00	0.10	0.36	0.46	0.09

图 A.49 总的流水线 CPI 和停顿的 4 种主要来源。影响 CPI 的主要因素是浮点结果停顿和转移停顿，其次是 load 和浮点资源停顿

A.7 相关问题

RISC 指令系统和流水线效率

我们已经讨论过使用简化指令系统来构建流水线的好处。简单的指令系统还提供了另外一个好处，即通过优化代码实现流水线的高效率变得相对比较简单。考虑下面的简单例子：假设我们需要将存储器中的两个数相加，并把结果存入存储器。在某些复杂指令系统中，只需要一条指令；在其他的指令系统中，则可能需要 2 条至 3 条指令。一个典型的 RISC 指令系统需要 4 条指令（2 条 load，1 条 add，1 条 store）。在没有流水线停顿的情况下，这些指令是很难在大多数流水线中顺序执行的。

使用 RISC 指令系统，单独的操作是多个可分离指令的组合。可以由编译器来进行调度（使用我们早先介绍的技术和在第 2 章中介绍的更强大的技术），或者使用动态硬件调度技术（将在下面和第 2 章中进行详细的介绍）。更高的效率，实现简单，所有这些优势很好地说明了一个问题：为什么近期的复杂指令系统的流水线在实现时都是先把复杂的指令转化为简单的类 RISC 指令，然后才进行流水线的调度和执行。第 2 章介绍了使用这种方法的 Pentium III 和 Pentium 4 处理器。

动态调度流水线

简单的流水线负责取指令并发射指令，除非在流水线中的指令与刚取到的指令之间存在数据相关，且不能通过旁路技术或直通技术避免。直通技术能够有效减少流水线的延迟时间，使得相关关系不会引起冒险。如果存在实在不可避免的数据相关，那么检测冒险的硬件将停止有相关的指令及其后面的指令进入流水线，相关被消除之前不会再取出和发射新的指令。为了减少性能损失，编译器将试图调度指令以避免冒险，这些就是通常所说的编译器调度法，也叫静态调度法。

几个早期的处理器还使用了另外一种方法——动态调度法。由硬件动态调整指令执行顺序以减少停顿的影响。本节通过介绍 CDC 6600 的记分板技术来说明动态调度法。读者将会发现，相对第 2 章中介绍的更复杂的 Tomasulo 算法来说，动态调度法的资料是比较容易找到的。先阅读这些资料可以更好地帮助读者理解后面的内容。

到目前为止,本附录中讨论的所有技术都采用按序的指令发射机制,即如果指令在流水线中被停顿,那么后继指令也无法前行。因此,若两条紧挨着的指令存在相关关系,就会引起流水线的停顿。

在MIPS流水线发展的初期,结构冒险和数据冒险均是在指令译码阶段(ID)进行检测的。若一条指令可以正确执行,才会从ID发射出去。为了使上例中的SUB.D指令能够开始执行,就必须把发射流水段分离成两个阶段:检测结构冒险和等待不存在数据冒险的情况。在发射指令时仍然要对结构冒险进行检测,仍然可以使用按序发射指令的方法,仍然可以要求指令在操作数准备好时就可以开始执行。这里的流水线采取的是乱序执行方式,指令的结束也是乱序的。在引入乱序执行这一机制之前,必须先弄清楚构成ID流水段的两个阶段:

1. 发射:译码指令,并检测结构冒险的情况。
2. 读操作数:等待直到不存在数据冒险,并读出操作数。

IF段在发射之前,读操作数之后是EX段,正如MIPS流水线中一样。在MIPS浮点流水线中,根据操作数的不同,有可能在执行阶段要用好几个周期,由此可能出现好几条指令同时并行执行的情形。所以需要区分指令什么时候开始执行,什么时候执行完毕。在两个时刻之间,就是指令的执行时间。这就使得多条指令可以同时被执行。除了流水线结构的这些改变之外,为了能够更好地开发利用更多的高级并行技术,还进行了如下一些变动:改变功能单元的数目、操作的延迟时间及功能单元的流水线机制。

采用记分板机制的动态调度法

在动态流水线中,所有的指令在发射阶段均是按序发射的,但在第二个节拍(读操作数阶段)有可能会被停顿或被旁路,从而开始乱序执行。记分板机制正是在资源单元充足、没有数据相关存在的前提下,允许指令乱序执行的一种技术。其命名取自于CDC 6600,该机器引入了记分板机制。

在考察记分板机制如何作用于MIPS流水线之前,还必须认识到,原先在MIPS浮点流水线和MIPS定点流水线中不存在的WAR冒险,指令的乱序执行也可能出现。考察下列代码:

DIV.D	F0, F2, F4
ADD.D	F10, F0, F8
SUB.D	F8, F8, F14

在指令ADD.D和SUB.D之间存在反相关:若流水线在执行ADD.D之前执行SUB.D,就违反了反相关关系而得出错误结果。同样,为避免发生输出相关,还必须对WAW冒险进行检测(如果SUB.D的目标寄存器换成F10)。以下将看到记分板机制如何停顿有反相关关系的指令的执行,从而避免发生这类冒险。

采用记分板机制的目的,是在没有结构冒险的前提下,使每一条指令尽可能早执行,以保持每一个时钟周期执行一条指令的速率,即当某条将要执行的指令被停顿时,其他没有相关于任何正在执行或停顿指令的指令应能够接着发射并执行。记分板机制就是负责实现指令发射和执行的,也负责所有冒险的检测。乱序发射机制要求能同时有多条指令在EX阶段中执行,这可以通过配置多个功能单元或流水功能单元加以实现。从流水线控制的意义上说,流水功能单元和多功能单元是两种基本一样的方法,在此不妨假设处理器配置了多功能单元。

CDC 6600有16个单独的功能单元,包括4个浮点单元、5个用于存储器访问的单元和7个用于定点操作的单元。在MIPS系统结构中,除浮点单元之外的其他功能单元引起的延迟时间均很小,

所以记分板机制也是基本上针对浮点单元的。在此假设有2个乘法器，1个加法器，1个除法单元及1个实现存储器访问、转移操作及定点运算等所有操作的定点单元，这比起CDC 6600虽然简单了许多，但还是可以在不需要太多细节与例子的情况下，就能够把记分板的工作机制阐述得清清楚楚。MIPS和CDC 6600均采用load-store结构，所以记分板机制在这两种处理器上是相同的。图A.50给出了它的基本系统结构。

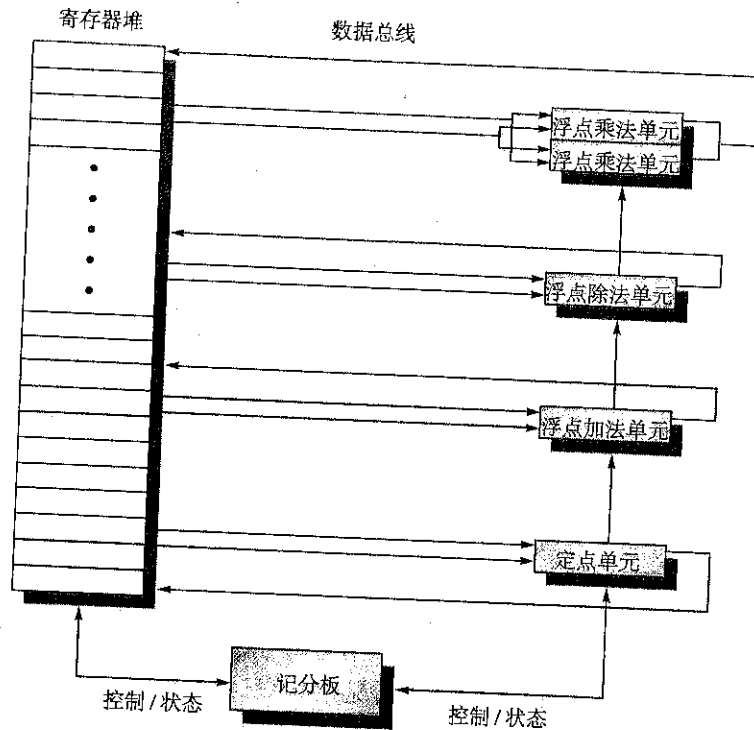


图 A.50 采用记分板机制的MIPS基本结构。记分板的功能是控制指令的执行（竖直的控制线）。所有数据通过总线（水平线，在CDC 6600中称干线）在寄存堆和功能单元间流动。有2个浮点乘法器、1个浮点除法器、1个浮点加法器和1个定点单元，且有一套总线（2输入1输出）为这一组功能单元提供服务。记分板机制的详细情况表示在图A.51至图A.54中

每条指令都要从记分板单元通过，并在这里建立相应的数据相关结构，这一步对应于MIPS流水线中的指令发射阶段，即代替了ID阶段的发射部分。然后由记分板单元决定何时可以读操作数及执行指令。如果判断该指令当前还不能立即执行，那么它会监视硬件上的每一个变化并决定该指令何时才能执行。记分板还可以控制指令写入目标寄存器的操作，这样，所有的冒险检测及解除工作都集中在记分板单元上。在看说明记分板单元的图A.51之前，我们首先考察流水线中发射和执行阶段是如何工作的。

每条指令在执行过程中经历4个阶段（主要看浮点操作数，因而不考虑涉及访问存储器的阶段）。接下来首先简单考察这几个阶段，并研究记分板是如何保存必需的信息并判断指令何时从一个阶段转到下一个阶段的。这4个阶段，相当于标准MIPS流水线中的ID，EX和WB，具体如下：

1. 发射：如果指令所用的功能单元正好空闲且没有其他活动的指令与它抢用同一个目标寄存器，那么记分板就把指令发射至该功能单元并更新自己内部的数据结构。这一阶段代替了

MIPS流水线中ID阶段的一部分。通过确保无其他活动的功能单元与其争用同一个目标寄存器,可以杜绝WAW冒险的出现,如果存在结构冒险或WAW冒险,就必须停止发射任何后继指令,直至冒险解除。当发射阶段被阻塞时,可能引起介于取指令阶段和发射阶段之间的缓存被充满。若缓存是单入口的,那么取指令段也会立即被停顿。如果缓存是一个可存储多条指令的队列,那么一旦队列充满就会引起停顿。

2. **读操作数:** 记分板监视操作数的可用情况,如果没有以前发射了的活动指令去写该操作数,或是存放该操作数的寄存器此刻正被某个活动的功能单元写入,那么源操作数是可用的。一旦源操作数可用,记分板便会通知相应功能单元读取该操作数并开始执行。通过这种方法,记分板解决了RAW冒险。指令从而可以乱序地进入执行阶段,这一阶段和发射阶段加起来相当于MIPS流水线中的ID阶段的功能。
3. **执行:** 功能单元在得到操作数之后立即开始执行。当执行结束,结果产生之后,便会通知记分板。这一阶段代替了MIPS流水线中的EX阶段。在MIPS浮点流水线中需要多个执行周期。
4. **写回结果:** 一旦记分板知道功能单元已经执行完毕,就会检查是否有WAR冒险,若有,则停顿完成的指令。

如前面所述的例子,指令ADD.D和SUB.D同时使用F8,从而造成WAR冒险。该例子的代码如下:

```
DIV.D    F0,F2,F4
ADD.D    F10,F0,F8
SUB.D    F8,F8,F14
```

F8是ADD.D指令的一个源操作数寄存器,也是SUB.D的目标寄存器,而且ADD.D还相关于前一条指令,这样,记分板就会停止SUB.D指令做写回结果的操作直至ADD.D读走其操作数。在通常情况下,如果存在以下情况,那么已经执行完成的指令不能做写回结果的操作:

- 在执行完的指令之前的指令尚未读取操作数,且其中一个操作数使用的寄存器正好与等待写入结果的寄存器相同。
- 某个操作数使用的寄存器正好是前面完成的指令结果写入的寄存器。

如果不存在WAR冒险,或冒险已经被解除,那么记分板就通知功能单元让其写回运算结果,这一阶段等同于简单MIPS流水线中的WB流水段。

从表面看起来,记分板似乎无法分出WAR冒险和RAW冒险,其实不然。

由于指令仅在两个源操作数已经在寄存器堆中准备好之时才去读取操作数,所以记分板没有使用相关直通技术,这也不会引起太大的恶性后果。与先前介绍的简单流水不同,指令一旦完成了执行阶段就令其尽可能早地把结果写回寄存器堆中(不存在WAR冒险),而不是去等待有可能还在几个周期之外的静态写入时钟的到来。这种影响是通过减少流水线延迟时间和利用直通来实现的。但是由于读取操作数阶段与写回结果阶段无法重叠执行,所以会出现一个周期的额外延迟时间,还必须使用缓存技术来消除这种开销。

基于其本身的数据结构,结合功能单元,记分板控制着指令执行阶段的前进状态,其操作是相当复杂的。因为只有较少几条源操作数总线和运算结果总线与寄存器堆相连,因而有出现结构冒险的可能。记分板必须保证有足够可用的功能单元和数据总线使指令由阶段2执行到阶段4。CDC

6600中把16个功能单元分成4组，每组提供一套数据总线（称为数据干线），且每一个时钟周期只允许每组中的一个单元读操作数或写运算结果。

下面来看带5个功能单元的MIPS记分板的数据结构。图A.51给出了执行以下例子时记分板中的信息分布：

```

L.D      F6,34(R2)
L.D      F2,45(R3)
MUL.D    F0,F2,F4
SUB.D    F8,F6,F2
DIV.D    F10,F0,F6
ADD.D    F6,F8,F2
  
```

指令状态									
指令		发射	读操作数			执行完成		写回结果	
L.D	F6,34(R2)	√	√			√		√	
L.D	F2,45(R3)	√	√			√			
MUL.D	F0,F2,F4	√							
SUB.D	F8,F6,F2	√							
DIV.D	F10,F0,F6	√							
ADD.D	F6,F8,F2								

功能单元状态									
名称	忙	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	是	Load	F2	R3				否	
Mult1	是	Mult	F0	F2	F4	Integer		否	是
Mult2	否								
Add	是	Sub	F8	F6	F2		Integer	是	否
Divide	是	Div	F10	F0	F6	Mult1		否	是

寄存器结果状态									
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1	Integer			Add	Divide			

图 A.51 记分板的组成。每一条已经发射或即将发射的指令都将进入指令状态表中，每一个功能单元也在功能单元状态表中对应一个状态。一旦一条指令被发射了，就会在功能单元状态表中保存一份操作数的记录。寄存器状态表则指出哪一个功能单元将产生结果，其表中的条目数等于寄存器的个数。上面的指令状态表指出：(1) 第一个L.D已经执行完成并写回了结果；(2) 第二个L.D已经执行完成但还没有写回结果。MUL.D, SUB.D, DIV.D均已发射但处于停顿状态，在等待各自的操作数。功能单元状态表则指出，第一个乘法单元正等待定点单元的结果，加法单元也在等待定点单元的结果，除法单元则等待着第一个乘法单元的结果。ADD.D指令由于结构冒险而被停顿，等SUB.D完成之后才能解除。记分板中某一个条目空则表示未被使用。例如，Rk字段在load操作时未被使用，Mult2单元未被使用，它们对应的字段因此也没有意义。当一个操作数被读走之后，其对应的Rj或Rk字段则会置成“否”。图A.54给出了最后一步重要的原因

记分板由3部分构成:

1. 指令状态表: 指示出指令所处的是4个阶段中的哪一个。
2. 功能单元状态表: 指出功能单元(FU)的状态, 每一个功能单元对应有9个字段。
 - Busy: 指示功能单元是否在使用。
 - Op: 使用该单元的操作符(如加或减)。
 - Fi: 目标寄存器。
 - Fj, Fk: 源寄存器。
 - Qj, Qk: 产生源寄存器Fj和Fk的功能单元。
 - Rj, Rk: 标志Fj和Fk是否准备好的标志位。
3. 寄存器结果状态表: 指出哪一个功能单元将写入这个寄存器, 如果没有指令写这个寄存器, 则这个字段置为空。

下面, 先看图A.51中的代码序列是如何连续执行的, 然后将详细讨论记分板控制执行的条件。

例题 先假定EX段浮点功能单元所需的延迟时间为加法2个时钟周期、乘法10个时钟周期、除法40个时钟周期。用图A.51所举的代码段及所假设的指令状态表示当指令MUL.D和DIV.D分别到达写回结果状态时记分板各状态表的情况。

解答: 从第二条L.D指令到MUL.D, ADD.D和SUB.D, 从MUL.D到DIV.D, 以及SUB.D到ADD.D间均存在RAW数据冒险。而从DIV.D到ADD.D和SUB.B则存在WAR数据冒险。ADD.D和SUB.B指令在使用加法功能单元时会产生结构冒险。图A.52和图A.53表示指令MUL.D和DIV.D分别到达写回结果阶段时各状态表的状况。

现在来看记分板工作的详细情况, 考察它是如何使各指令正常执行的。图A.54给出记分板对各条指令继续运行的要求, 及当指令可以继续执行时需要记录的必要信息。记分板就像本章中所碰到的其他一些结构那样记录操作数的具体信息, 如寄存器序号等。例如, 当一条指令被发射时, 必须记下其源寄存器。由于我们通常使用Regs[D]来表示一个寄存器的值, 而D是该寄存器的名字, 所以不会引起二义性。如Fj[FU]←S1表示把S1所指的寄存器名称放到Fj[FU]中, 而不是把S1中的值放到Fj[FU]中。

记分板技术的优点和代价是一个值得探讨的问题。CDC 6600的设计者测得使用这项技术之后流水线的性能对FORTRAN程序提高1.7倍, 比手写汇编代码提高2.5倍。但是, 这一结果是在采用了软件流水调度、半导体主存及Cache等技术出现之前测得的。记分板单元在CDC 6600中的硬件逻辑复杂度大体等同于一个功能单元, 成本很低。其耗费的成本主要在大量的数据总线上, 所需成本是处理器按顺序执行指令时, 或每一个执行周期只初始化一条指令时的4倍。近年来人们尝试采取一个时钟周期发射多条指令去抵消多总线引起的高成本, 这也促使动态调度方法引起了人们更多的兴趣。同样, 基于动态调度的预测方法(在4.7节中阐述)的出现也促使动态调度方法有更大的发展。

记分板运用指令级并行技术减少程序中数据相关关系所造成的停顿。在消除流水线停顿时, 记分板会受到如下几个因素的限制:

1. 指令中可开发的并行性: 这决定了有多少独立的指令能够被发掘出来并行执行。如果每条指令均相关于前一条指令, 那么没有什么动态调度方法可以减少这一流水线的停顿。如果这些指令必须从同一个基本块中同时被抽取出来, 那么对流水线的影响是相当严重的。

指令	指令状态			
	发射	读操作数	执行完成	写回结果
L.D F6,34(R2)	√	√	√	√
L.D F2,45(R3)	√	√	√	√
MUL.D F0,F2,F4	√	√	√	√
SUB.D F8,F6,F2	√	√	√	√
DIV.D F10,F0,F6	√	√	√	√
ADD.D F6,F8,F2	√	√	√	√

功能单元状态								
名称	忙	Op	Fi	Fj	Fk	Qj	Qk	Rj Rk
Integer	否							
Mult1	是	Mult	F0	F2	F4			否 否
Mult2	否							
Add	是	Add	F6	F8	F2			否 否
Divide		Div	F10	F0	F6	Mult1		否 是

寄存器结果状态								
	F0	F2	F4	F6	F8	F10	F12	... F30
FU	Mult 1			Add		Divide		

图 A.52 MUL.D 指令到达写回结果阶段时，记分板各状态表的信息。此时 DIV.D 尚未读取任何操作数，因为它相关于乘法结果。指令 ADD.D 已经读取操作数并正在运算中，但它需要 SUB.D 的运算结果。由于 ADD.D 写回运算结果的寄存器 F6 与 DIV.D 相关，因而也不能执行写回结果操作。Q 字段仅在一个功能单元等待另一个功能单元时才相关。

2. 记分板单元的入口数：这影响到流水线可以提前查找独立不相关指令的能力。如果把作为候选进入执行状态的待检查的指令系统称为窗口，记分板的尺寸大小即决定了窗口的大小。在本节中，始终假设窗口大小不会超越一个转移，即窗口中始终包含的是一个简单基本块中的线性代码。在第2章中将会给出窗口超出一个转移的情况。
3. 功能单元的数量与类型：这影响到结构冒险的可能性。在采用动态调度方法时，冒险的可能性也会增加。
4. 反相关和输出相关的存在：这些都会导致 WAR 冒险和 WAW 冒险。

第2章和第3章集中讨论了如何更好地开发和利用指令级并行性技术。上面的2和3两个不利因素可以通过增加记分板的大小及功能单元的数量予以解决。然而，这些改变会增加复杂性并影响周期时间。WAW 冒险和 WAR 冒险在采用动态调度方法的处理器中显得更为突出一些，因为动态调度的流水线会产生更多的名字相关。如果在动态调度方法中，还采用多个迭代互相重叠执行的转移预测技术，则将使 WAW 冒险显得更加突出。

指令		指令状态			
		发射	读操作数	执行完成	写回结果
L.D	F6,34(R2)	√	√	√	√
L.D	F2,45(R3)	√	√	√	√
MUL.D	F0,F2,F4	√	√	√	√
SUB.D	F8,F6,F2	√	√	√	√
DIV.D	F10,F0,F6	√	√	√	√
ADD.D	F6,F8,F2	√	√	√	√

功能单元状态									
名称	忙	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	否								
Mult1	否								
Mult2	否								
Add	否								
Divide	是	Div	F10	F0	F6			否	否

寄存器结果状态									
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU									Divide

图 A.53 DIV.D到达写回结果阶段时记分板各状态表的信息。由于DIV.D已经读取了寄存器 F6，因而 ADD.D 得以执行写回结果操作，这时只剩下 DIV.D 尚未执行完成

指令状态	须满足的条件	记录工作
发射	功能单元不忙，且没有结果写目标寄存器	$Busy[FU] \leftarrow yes; Op[FU] \leftarrow op; Fi[FU] \leftarrow D;$ $Fj[FU] \leftarrow S1; Fk[FU] \leftarrow S2;$ $Qj \leftarrow Result[S1]; Qk \leftarrow Result[S2];$ $Rj \leftarrow not Qj; Rk \leftarrow not Qk; Result[D] \leftarrow FU;$
读操作数	Rj 和 Rk	$Rj \leftarrow No; Rk \leftarrow No; Qj \leftarrow 0; Qk \leftarrow 0$
执行完成	功能单元已执行完成	
写回结果	$\forall f((Fj[f] \neq Fi[FU] \text{ or } Rj[f] = No) \& (Fk[f] \neq Fi[FU] \text{ or } Rk[f] = No))$	$\forall f(\text{if } Qj[f]=FU \text{ then } Rj[f] \leftarrow Yes);$ $\forall f(\text{if } Qk[f]=FU \text{ then } Rk[f] \leftarrow Yes);$ $Result[Fi[FU]] \leftarrow 0; Busy[FU] \leftarrow No$

图 A.54 执行阶段每一步需要做的检查及记录工作。其中FU表示指令所使用的功能单元，D是目标寄存器名，S1和S2表示源寄存器名，Op表示将执行的操作。用Fj[FU]表示功能单元FU对应的Fj中的信息。Result[D]表示寄存器D中的值。如果另一条指令使用了写回结果的条件测试指令的目标操作数(Fi[FU])作为其源操作数(Fj[f]或Fk[f])，且其他指令已经执行了写入寄存器的操作(Rj = Yes或Rk = Yes)，则写回结果的条件测试指令可以避免有WAR冒险时的误写操作。变量f用于任何功能单元

A.8 谬误和易犯的错误

谬误：不可预测的执行序列可能产生无法预测的冒险。

我们首先可能想到的是，WAW冒险永远不会发生，因为没有哪个编译器会产生对同一个寄存器的两次写，却没有在中间进行读。但是当指令序列不可预测时，WAW冒险是可能发生的。例如，编译器认为一个转移不会被选中，于是第一次写发生在延迟槽内，但实际上这个转移被选中了，下面就是这样的一个指令序列：

```

      BNEZ    R1,foo
      DIV.D   F0,F2,F4; 从转移不成功路径移至延迟槽
      .....
      .....
foo:    L.D    F0,qrs

```

如果转移被选中，在DIV.D指令执行完成之前，L.D指令有可能到达WB段，于是引起了WAW冒险。硬件必须检测到这种冒险，而且可能要停顿L.D指令的发射。另一种可能引起WAW冒险的情况是第二次写发生在陷阱服务程序中。当发生这种情况时，一条需要写结果的陷阱指令继续执行直到完成，而在陷阱服务程序里也要写同一个寄存器。硬件也必须检测并防止发生这种情况。

易犯的错误：扩展流水线可以影响设计的其他方面，从而提高性价比。

对此最有说服力的是VAX的两个机型：8600和8700。当8600刚上市时，时钟周期的长度为80 ns。随后，另一个型号8650也上市了，它的时钟周期是55 ns。8700有一个简单的微指令级流水线，采用体积小一些的CPU，它的时钟周期为45 ns。总的比较是8650在CPI方面领先大约20%，8700在时钟频率方面领先约20%，即8700用较少的硬件获得了基本相当的性能。

易犯的错误：用未经优化的代码评价静态调度和动态调度的性能。

未经优化的代码包含有冗余的load, store和其他可以被优化去掉的多余操作，这种代码远比经过调度的高效率代码更容易调度，对于控制冒险和RAW冒险都是如此。R3000与MIPS有几乎相同的流水线，当运行gcc时，经过调度的代码比未经调度的代码多18%的空操作周期。实际上，当然是经过优化的程序运行得更快，因为它的指令少。所以进行评价时一定要以经过优化的代码为依据，因为在实际的系统中你还能从调度之外获得别的优化，因而经过优化的代码更能反映真实情况。

A.9 结论

20世纪80年代初，流水线技术是超级计算机和价值数百万美元的大型机使用的专有技术。20世纪80年代中期，出现了第一个使用流水线技术的微处理器，从此，微机使用流水线技术成为可能。20世纪90年代初，高端嵌入式系统CPU开始使用流水线技术、复杂动态调度方法和多发射方法（在第2章中介绍）。本附录介绍的资料在本书第一版出版时是先进的，但是现在这些技术已经在不足10美元的处理器中广泛使用了。

A.10 历史回顾和参考文献

随书光盘上的K.4节介绍了流水线和指令级并行的发展历程。我们为深入阅读和探讨这些主题提供了参考文献。

附录B 指令系统原理与实例

- A n 把存储器地址 n 里面的数字加到累加器中。
E n 如果累加器中的数字大于或等于 0 就执行下一条指令，否则顺序执行。
Z 使机器停止运行并响铃。

Wilkes 和 Renwick, 选自 18 Machine Instruction for the EDSAC (1949)

B.1 简介

本附录主要讨论指令集系统结构,即计算机对程序员和编译器开发者可见的部分。这部分内容主要为读者提供一个复习材料,并将其作为相关背景的介绍。本附录介绍现有的各种指令集系统结构设计方案,主要包括四个方面:首先,介绍一种指令系统分类的方法,并对各种指令系统设计方案的优缺点进行定性评价;其次,介绍并分析几种有别于特定指令系统的指令系统评价方法;然后讨论有关计算机语言、编译器以及这两者与指令集系统结构关系的相关问题;最后在“综合”小节中介绍典型的 RISC 系统结构——MIPS 指令系统是如何体现这些思想的。最后对指令系统设计中出现的谬误与易犯的的错误进行总结。

为了更好地阐述原理,在附录 J 中还将介绍四种通用的 RISC 系统结构 (MIPS, Power PC, Precision Architecture 和 SPARC) 和四种嵌入式 RISC 处理器 (ARM, Hitachi SH, MIPS 16 和 Thumb), 以及三个原有的系统结构 80x86, IBM 360/370 和 VAX。在讨论如何对各种系统结构进行分类之前,首先介绍一下指令系统的评价方法。

整个附录讨论了多种不同指令集系统结构的评价方法。显然,评价的结果依赖于被评价的程序以及所使用的编译器。评价的结果并不是绝对的,如果使用不同的编译器或不同的评价程序,其结果可能会有些差别。本书提到的各种评价方法具有一定的代表性,代表了一些典型的程序。许多评价方法都使用一个基准测试程序集给出,这样能够较好地显示结果,还可以看出不同程序之间的差别。新型计算机的设计者在做出系统结构设计的决策前,可能要分析大量的程序评价结果。这通常采用动态的评价方法,即评价事件的出现频率需要通过评价程序执行过程中该事件发生的次数进行加权处理。

在开始介绍基本原理之前,首先回顾一下第 1 章中提到的三个应用领域。桌面计算注重程序的定点和浮点运算性能,而不太考虑程序的大小以及处理器的功耗。例如,五代 SPEC 基准测试程序均不报告代码量的大小。服务器现在主要应用于数据库、文件服务器、Web 应用以及一些多用户分时应用。因此,尽管所有服务器的处理器都支持浮点指令,但浮点运算性能的重要性远不及定点和字符串方面的处理性能。嵌入式应用注重成本和功耗,所以代码量的大小是至关重要的,因为存储器越小就意味着成本更低,功耗更小。为了降低芯片的成本,一些指令(比如浮点指令)则成为可定制的选项。

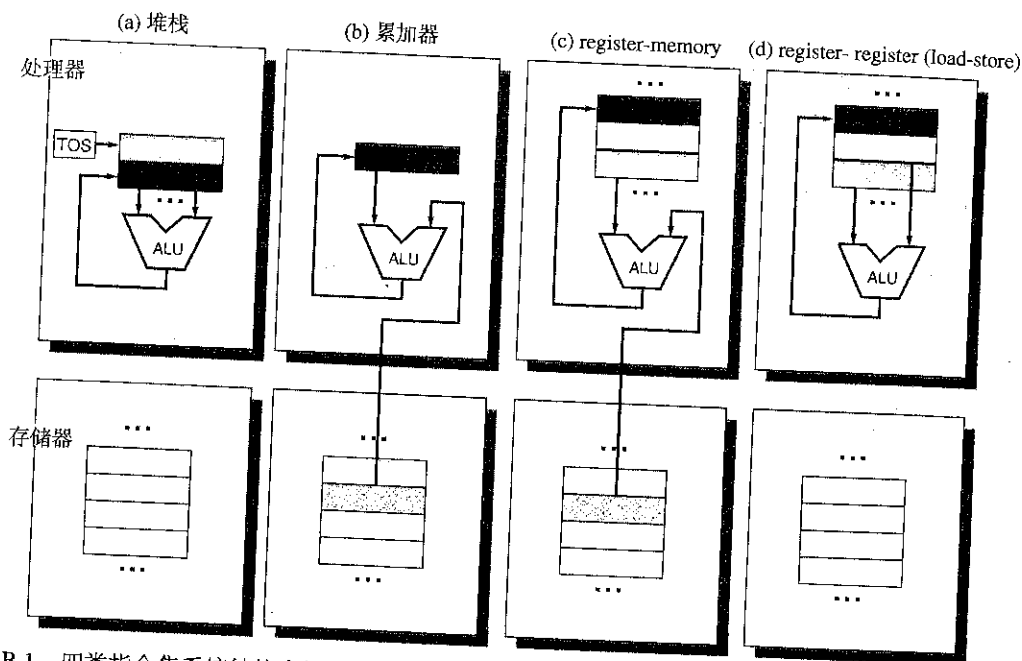
如此看来,指令系统在这三个应用领域的情况很相似。实际上贯穿本附录的 MIPS 系统结构在桌面、服务器及嵌入式应用中均有广泛应用。

80x86是一个与RISC有很大区别且非常成功的系统结构(见附录J),然而它的成功丝毫没有掩盖掉RISC指令系统的优点。对PC软件二进制兼容的商业重要性,以及在摩尔定律的影响下晶体管容量的不断增加,促使Intel在内部使用RISC指令系统,同时对外支持80x86指令系统。近来的80x86多处理器,像Pentium 4,其硬件将80x86指令转换成类RISC指令,然后在芯片内部执行这些转换后的指令。在程序员看来这是80x86的系统结构,而同时基于性能考虑,计算机设计者可以实现RISC风格的处理器。

至此,我们已经具备了背景知识。下面首先介绍指令集系统结构的分类。

B.2 指令集系统结构的分类

指令集系统结构最根本的区别在于处理器内部的存储类型,所以本节主要讨论这方面的内容。主要的设计方案包括使用堆栈、累加器或者一组寄存器。操作数可以显式指定或隐含指定:堆栈系统结构中操作数隐含地位于栈顶,累加器系统结构中的一个隐含操作数就是累加器。通用寄存器系统结构中只能明确地指定操作数,不是寄存器就是存储器地址。图B.1所示为这些不同的系统结构的方块图,图B.2说明了代码 $C = A + B$ 在这三类系统结构中分别是如何表示的。显式指定的操作数有的通过直接访问存储器来读取,有的需要先载入到临时存储器里面才能访问,这由指令集系统结构的类型和所选的特定指令来决定。



图B.1 四类指令集系统结构中操作数的位置。箭头指示操作数是输入还是ALU运算的结果,或者既是输入也是结果。灰度较轻的表示输入,较重的表示结果。在(a)中,栈顶寄存器(TOS)指向堆栈顶部的输入操作数,并与下面的操作数结合在一起。第一个操作数被从堆栈中删除,运算结果存放在第二个操作数的位置,同时栈顶寄存器指向运算结果。所有的操作数都是隐含的。在(b)中,累加器既是隐含的输入操作数也是运算结果。在(c)中,其中一个操作数在寄存器中,另一个在存储器中,运算结果存放在寄存器中。在(d)中,所有的操作数都是寄存器,与堆栈结构类似,也只能通过一些单独的指令传输到存储器中:(a)中是push或pop,(d)中是load或store

堆栈	累加器	寄存器 (register-memory)	寄存器 (load-store)
Push A	Load A	Load R1, A	Load R1, A
Push B	Add B	Add R3, R1, B	Load R2, B
Add	Store C	Store R3, C	Add R3, R1, R2
Pop C		Store R3, C	

图 B.2 四种指令系统执行 $C = A + B$ 的指令序列。在堆栈结构和累加器结构中加法指令的操作数是隐含的,而在寄存器结构中操作数必须明确指定。假设 A, B 和 C 都在存储器中且 A 和 B 的值不会破坏。图 B.1 所示为不同系统结构下的加法操作

正如图 B.1 和图 B.2 所示,按照寄存器访问方式划分,有两种寄存器系统结构的计算机:一种称为 **register-memory 系统结构**,任何一条指令都可以访问存储器;另外一种称为 **load-store 系统结构**,只能通过 load 和 store 指令来访问内存。还有一种现实中不存在的结构,即 **memory-memory 系统结构**,就是把所有的数据都保存在存储器中。一些指令集系统结构除累加器之外还有别的寄存器,但是对它们的使用进行了一些限制。这种结构有时称为**扩展累加器**或**专用寄存器**计算机。

虽然大多数早期的机器使用堆栈或者类似于累加器型的系统结构,但实际上 1980 年之后设计的计算机都使用 load-store 寄存器系统结构。通用寄存器 GPR 出现的主要原因有两个:首先,寄存器同其他的处理器内部存储结构一样,但比存储器快;其次,编译器使用寄存器很方便,比使用其他的内部存储形式效率更高。例如,在一台寄存器系统结构的计算机上,表达式 $(A \times B) - (B \times C) - (A \times D)$ 可以按任意顺利来执行乘法,因为操作数的位置不同,或者由于应用流水线的原因(见第 3 章),可以采用效率最高的顺序来执行这三个乘法。但是在堆栈计算机上则只有一种计算顺序,因为操作数隐含在堆栈中,且必须多次载入。

更重要的是,寄存器可以用来存放变量。把变量加载到寄存器里面可以减少数据流量,加速程序运行(因为寄存器比存储器要快),还可以改善代码密度(因为指明一个寄存器比指明一个存储器地址所需的位数要少)。

正如 B.8 节将讲到的,编译器开发者希望所有的寄存器都相同,而且都不是保留用于特殊用途的。早期的计算机在这种构想和保留一些寄存器用于特殊用途的设计之间进行折中,结果是减少了通用寄存器的数量。如果通用寄存器的实际数量太少,把变量放在寄存器中就没有实际意义,这样,编译器只能将所有没有分配的寄存器都用于表达式计算。

多少个寄存器才够用呢?这个问题的答案显然取决于编译器如何利用这些寄存器。大多数编译器都保留一部分寄存器来进行表达式计算,一部分用来传递参数,剩下的允许用来存放变量。现代的编译技术及其有效使用较大数量寄存器的能力导致了寄存器数目在近来的系统结构中不断增加。

通用寄存器系统结构主要由指令系统的两个特性来划分。这两个特性都与典型的算术或者逻辑指令(ALU 指令)操作数的性质有关。第一个特性即指令是否有两个或者三个操作数。在三个操作数的格式中,指令包含一个结果操作数(目的操作数)和两个源操作数。在两个操作数的格式中,有一个既是结果操作数也是源操作数。第二个特性是指在 ALU 指令中可以有多少个操作数作为存储器地址。典型的 ALU 指令中所支持的存储器操作数的数量可能是从 0 到 3 个不等。图 B.3 用几个计算机的例子说明了这两条特性。虽然有七种可能的组合,但现实中几乎所有的计算机都可以归为三类。正如前面提到的,这三种是 register-register (load-store), register-memory 和 memory-memory。

每一种方案的优缺点都列在图 B.4 里面。当然,这些优缺点不是绝对的:首先这些都是定性的评价,其次实际效果与编译器和实现的方法有关。编译器可以很容易地把使用 memory-memory 指令的 GPR (通用寄存器) 计算机当做一个 load-store 计算机来处理。指令编码和执行一个任务所需

要的指令条数是对系统结构影响最大的因素之一。我们将会在附录A和第2章看到这几种系统结构方案对实现方法的影响。

存储器地址个数	最多操作数个数	系统结构类型	举例
0	3	Load-store	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, TM32
1	2	Register-memory	IBM360/370, Intel 80x86, Motorola 68000, T1 TMS320C54x
2	2	Memory-memory	VAX(也有 3 个操作数的格式)
3	3	Memory-memory	VAX(也有 2 个操作数的格式)

图 B.3 所列计算机每条典型ALU指令中的操作数总数和存储器操作数的组合情况。在ALU指令中不对存储器进行操作的计算机称为load-store或者register-register计算机。ALU指令中有一个存储器操作数的指令称为register-memory指令, 有多个存储器操作数的指令称为memory-memory指令

类型	优点	缺点
Register-register (0, 3)	简单、定长的指令编码; 简单的代码生成模式; 每条指令运行的时钟周期相近 (详见附录A)	指令数比可以直接访问存储器的系统结构多; 指令多和指令密度低使程序变得很大
Register-memory (1, 2)	数据不需要专门的载入指令就可以直接访问; 指令格式更加易于编码, 代码密度高	由于源操作数在二元操作中被破坏了, 所以操作数不是等价的; 在一条指令里面同时对存储器地址和寄存器号码进行编码会限制寄存器的数量; 操作数位置不同使得每条指令执行所需的时钟周期不同
Memory-memory (2, 2)或(3, 3)	最紧凑。不浪费寄存器来做临时交换空间	指令长短各不相同, 特别是三操作数指令; 同样, 每条指令的操作各不相同; 存储器访问带来了存储器瓶颈

图 B.4 三种最常见通用寄存器计算机的优缺点。符号 (m, n) 表示 n 个操作数中有 m 个存储器操作数。一般而言, 计算机可能的选择方案越少, 编译器要做的选择越少, 编译器的工作越简单 (见B.8节)。有大量灵活指令格式的计算机减少了用来进行编码的位数。寄存器的数量也会影响到指令的长短, 因为每个寄存器需要1b (寄存器数) 位来标志。因此在register-register系统结构中将寄存器数加倍, 指令就需要多用3位, 这在32位指令系统中大约占10%

总结: 指令集系统结构分类

在本节和B.3节到B.8节的结尾部分对将在一个新的指令集系统结构中遇到的一些特性进行了总结, 为在B.9节中介绍MIPS系统结构奠定了基础。从下一节开始, 将讨论通用寄存器的使用。结合图B.4和附录A, 可以对通用寄存器系统结构的load-store版本做一些展望。

以上介绍了系统结构的分类, 下一节将要介绍操作数寻址。

B.3 存储器寻址

不管所采用的系统结构是register-register, 还是需要访问存储器得到操作数的系统结构, 总是需要定义一套如何指定存储器地址、如何对地址进行解释的规则。本节讨论的评价方法, 多半与具体机器无关。在某些情况下, 编译技术对评价结果有很大的影响。这些评价方法都使用了优化编译器, 因为编译技术起着很重要的作用。

存储器地址表示

存储器地址是如何表示的呢？也就是说，以存储器的地址和长度来访问的对象是什么？本书讨论的所有指令系统（某些 DSP 指令系统除外）都是字节寻址的，都提供了字节（8 位）、半字（16 位）和字（32 位）寻址。大多数计算机还提供了双字（64 位）寻址。

在较大的数据对象中字节的顺序有两种不同的约定。小端模式把地址为“x...x000”的字节放在整个字的最低有效位置上，字节内各位编码为

7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---

大端模式把地址为“x...x000”的字节放在整个字的最高有效位置上。字节内各位编码为

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

在同一台计算机内部进行操作时，字节顺序往往被忽略，只有同时用字节方式和字方式访问同一位置时才有差别，但是当在不同字节顺序的机器之间交换数据时，就特别需要注意字节顺序。在“低字节优先”情形下进行字符串比较就不能匹配字符串的正常顺序，例如“backwards”在寄存器中是“sdrawkcab”。

在许多计算机上，对大于一个字节的数据的寻址必须**对齐**。假设一个大小为 s 个字节的数据地址 A ，如果 $A \bmod s = 0$ ，访问该地址就是对齐的。图 B.5 说明了什么样的地址是对齐的，什么样的地址是未对齐的。

字节地址的低3位								
数据宽度	0	1	2	3	4	5	6	7
1字节（字节）	对齐	对齐	对齐	对齐	对齐	对齐	对齐	对齐
2字节（半字）	对齐		对齐		对齐		对齐	
2字节（半字）		未对齐		未对齐		未对齐		未对齐
4字节（字）	对齐				对齐			
4字节（字）		未对齐				未对齐		
4字节（字）		未对齐					未对齐	
4字节（字）		未对齐						未对齐
8字节（双字）	对齐							
8字节（双字）		未对齐						
8字节（双字）		未对齐						
8字节（双字）		未对齐						未对齐
8字节（双字）		未对齐						
8字节（双字）		未对齐						
8字节（双字）		未对齐						
8字节（双字）		未对齐						

图 B.5 按字节编址的计算机中字节、半字、字和双字的对齐与不对齐。如果数据没有对齐，它们的存取需要对存储器进行两次访问才能完成。如果一个数据是对齐的，只要存储器与其大小一致，那么这个数据就可以通过一次访问存储器完成。上图显示了以 8 个字节进行组织的存储器，标志每一列的位移量指定了地址的低 3 位

设计计算机时,为什么要有对齐的限制呢?由于存储器都是字或者双字整数倍对齐的,不对齐会导致硬件实现的复杂性。一次不对齐的存储器访问会导致多次对齐存储器访问。因此,即使是在没有对齐限制的计算机里面,对齐访问的程序也会运行得比较快。

即使数据是对齐的,如果要支持字节访问、半字访问和字访问,仍需要一个对齐网络来对齐64位寄存器里面的字节、半字和字。例如,在图B.5中,假如要读取一个字节,地址低3位的值为4,那么必须右移3字节使其与64位寄存器的合适位置对齐,根据指令的不同,计算机或许还要对其值进行符号扩展。因为只有被影响到的字节才会被改变,所以存储起来则很容易。在一些计算机上,字节、半字和字操作不会影响到寄存器的高位部分(虽然本书中提及的所有计算机都允许字节和半字节访问,但是只有IBM360/370, Intel 80x86和VAX系统计算机支持小于一个字长度的ALU操作)。

在讨论了各种存储器地址表示后,下面继续讨论指令如何确定存储器地址,即寻址方式。

寻址方式

现在只要给出一个地址,就知道要访问存储器中的哪些字节。本节将讨论寻址方式——各种系统结构如何指定它们所要访问的对象的地址。寻址方式指定常量、寄存器和存储器位置。当对一个存储器位置被占用时,由这种寻址方式所指定的实际存储器地址称为有效地址。

图B.6列出了近期的计算机中使用的所有数据寻址方式。立即数或直接操作数通常也被认为是一种存储器寻址方式(尽管它们要访问的数值在指令流里),然而,寄存器却常常被分离出来,我们把依赖于程序计数器的PC相对寻址也分离出来。PC相对寻址主要在控制转移指令中用来指定代码位置。控制指令中PC相对寻址的使用将在B.6节详细讨论。

不同系统结构中寻址方式的名称各不相同,图B.6给出了最常见的名称。本书中使用C语言的一种扩展来描述硬件。该图中只用了一个不属于C语言的特性:左箭头(\leftarrow)用来赋值。数组Mem表示主存储器位置的名字,数组Reg表示寄存器的名称。所以,Mem[Regs[R1]]指的是由寄存器R1的内容指定的存储器地址的内容。后面会讨论如何存储和传输小于一个字长度的数据。

寻址方式能够明显减少指令数量,但是这也增加了设计计算机的复杂度和实现这些寻址方式的计算机的平均CPI(每条指令的时钟周期数)。因此,对系统结构设计者来说,如何选择各种寻址方式是很重要的。

图B.7给出了VAX系统结构的计算机上用三个程序所测试出的寻址方式的结果。在本附录里面,之所以使用旧的VAX系统结构计算机来进行测试,是因为它有丰富的寻址方式,且对存储器寻址的限制很少。例如,它支持图B.6中列出的所有模式。在本附录的大多数测试中,我们使用更多的近期register-register系统结构说明程序如何使用现代计算机的指令系统。

正如图B.7所示,立即数寻址和位移量寻址是用得最多的寻址方式。下面介绍这两种经常使用的模式的一些特性。

位移量寻址方式

位移量寻址方式的主要问题是位移的范围。根据不同的位移量大小,可以决定到底使用多长的位移量。选择位移量的长度是很重要的,因为它直接影响到指令的长度。图B.8列出了在运行三个测试程序数据访问的load-store结构计算机上所得到的测试结果。关于转移的问题将会在B.6节讨论。数据访问和转移有很大差别,把它们放在一起讨论没有什么意义。

寻址方式	指令举例	含义	何时使用
寄存器寻址	Add R4, R3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Regs}[R3]$	数值在寄存器中
立即数寻址	Add R4, #3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + 3$	数值是常量
位移量寻址	Add R4, 100(R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[100 + \text{Regs}[R1]]$	存取局部变量(+ 模拟寄存器间接、直接寻址)
寄存器间接寻址	Add R4, (R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[\text{Regs}[R1]]$	使用指针或者计算出的地址进行寻址
间接寻址	Add R3, [R1 + R2]	$\text{Regs}[R3] \leftarrow \text{Regs}[R3] + \text{Mem}[\text{Regs}[R1] + \text{Regs}[R2]]$	有时用在数组寻址中:
索引寻址	Add R1, (1001)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[1001]$	R1是数组的基址, R2是索引值用来存取静态数据; 地址常量可能需要很大
存储器间接寻址	Add R1, @(R3)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Mem}[\text{Regs}[R3]]]$	如果 P3 是指针 p 的地址, 那么就得到 *p
自动递增寻址	Add R1, (R2) +	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]]$ $\text{Regs}[R2] \leftarrow \text{Regs}[R2] + d$	用在循环中递增变量。R2是数组的起始地址, 每次增加 d
自动递减寻址	Add R1, -(R2)	$\text{Regs}[R2] \leftarrow \text{Regs}[R2] - d$ $\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]]$	和自动递增类似, 自动递增/递减用来实现类似堆栈的 push/pop 功能
比例寻址	Add R1, 100(R2)[R3]	$\text{Regs}[R1] \leftarrow \text{Regs}[R12] + \text{Mem}[100 + \text{Regs}[R2] + \text{Regs}[R3]*d]$	用来对数组寻址。一些计算机可以用任意的索引(间接)寻址方式

图 B.6 寻址方式的例子、含义和使用。在自动递增/递减和比例寻址方式中, 变量 d 表示被存取数据项的大小(例如, 指令访问 1 个字节、2 个字节、4 个字节还是 8 个字节); 这意味着这些寻址方式只在被存取单元在存储器中相邻的情况下才有用。RISC 计算机中用位移量寻址模拟寄存器间接寻址和直接寻址; 如果位移量为 0 则位移量寻址模拟寄存器间接寻址, 如果寄存器为 0 则位移量寻址模拟直接寻址。在测试中, 使用每种模式的第一个名字。使用一种 C 语言的扩展来描述硬件

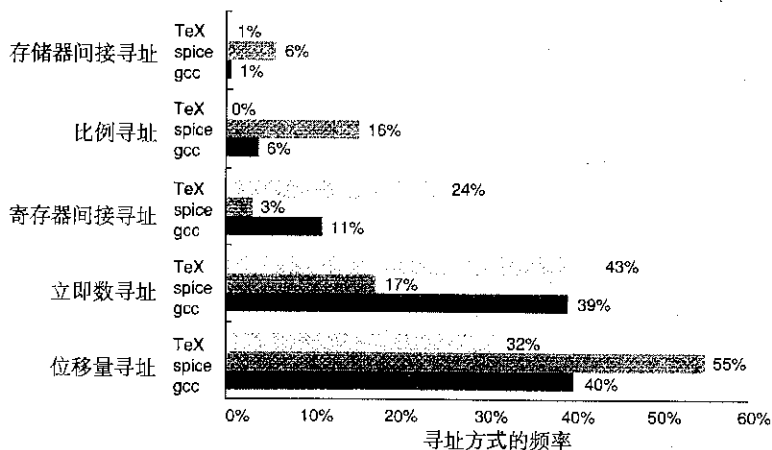


图 B.7 存储器寻址方式(包括立即数寻址)的使用总结。97%~100% 的存储器访问都使用这些主要寻址方式。寄存器模式未被统计在内, 但是事实上它占了访问操作数的一半, 而存储器模式(包括立即数寻址)占了另一半。当然编译器也影响到寻址方式的使用(见 B.8 节)。在 VAX 上存储器间接寻址可以使用位移量寻址、自动递增寻址和自动递减寻址来构成初始存储器地址。在这些程序中, 几乎所有的存储器间接寻址都基于位移量寻址方式。位移量模式包括所有的位移量长度(8, 16 和 32 位)。这里没有包括只用在分支指令中使用的 PC 相对寻址方式。本图只列出了平均使用频率超过 1% 的寻址方式

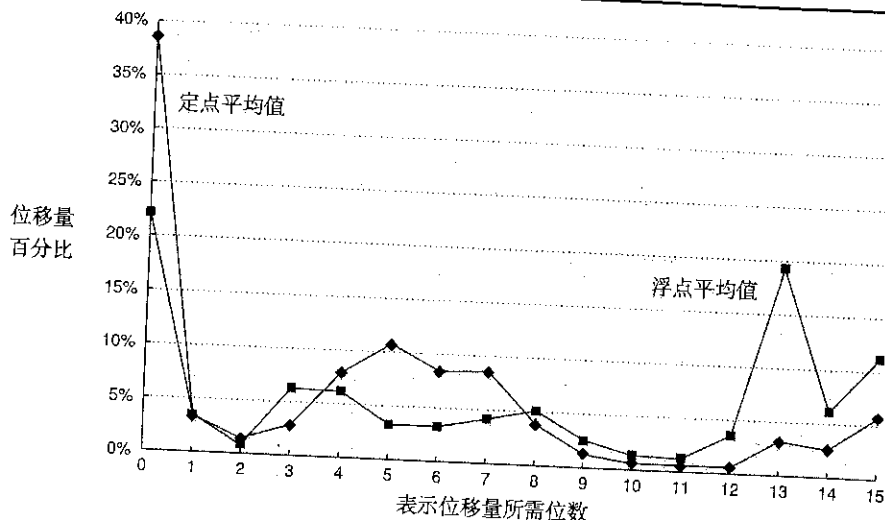


图 B.8 位移量值分布范围很广。虽然很多值比较小，但是还是有一些比较大的值。由于变量的存储位置和存取变量方式的不同（见 B.8 节）以及编译器使用的整个寻址方式的原因，位移量值分布范围很广。 x 轴是位移量以 2 为底的对数，也就是用来表示位移量范围所需数值的大小。 x 轴上的 0 位置表示位移量为 0 的部分所占的百分比。该图仅仅说明位移量的大小而不包括符号位，存储系统的设计对位移量的符号位有很大影响。绝大多数位移量是正的，但是最大位移（14 和大于 14 的）中很多是负的。由于这些数据是在一台 16 位位移量的计算机上得到的，我们不能从中得到使用更长位移量的数据访问的任何信息。这些数据是在为 SPEC CPU2000 进行了完全优化的 Alpha 结构计算机上测试的，包括定点程序（CINT2000）的平均值和浮点程序（CFP2000）的平均值。

立即数寻址方式

立即数寻址可以用于算术运算、比较指令（主要是分支转移指令）以及寄存器存取常数的指令。其中最后一种情况在把常数写进代码中出现，这些常数往往比较小，也有可能是很大的地址常量。对于立即数的使用，决定它们到底是需要支持所有操作还是只支持一部分操作是很重要的。图 B.9 中的统计图表列出了一个指令系统中常见定点操作使用立即数的频率。

另外一个重要的指令系统衡量标准是立即数的取值范围。与位移量相同，立即数的数值大小会影响指令的长度。正如图 B.10 所示，小立即数是最常用的。有时也使用大立即数，特别是在寻址计算中。

总结：存储器寻址

首先，由于这些技术的普遍使用，我们希望一个新的系统结构最少应该支持以下寻址方式：位移量寻址、立即数寻址和寄存器间接寻址。图 B.7 表明它们在测试所用到的寻址方式中占了 75%~99%。其次，位移量寻址方式中位移量的大小至少要有 12 到 16 位。根据图 B.8 中的数据，这样将覆盖 75%~99% 的位移量。最后，立即数数值的大小应该不少于 8 位到 16 位。

在学习了指令系统的分类、对 register-register 系统结构进行了评价，加上前面对寻址方式给出了建议之后，下面开始研究数据的大小和含义。

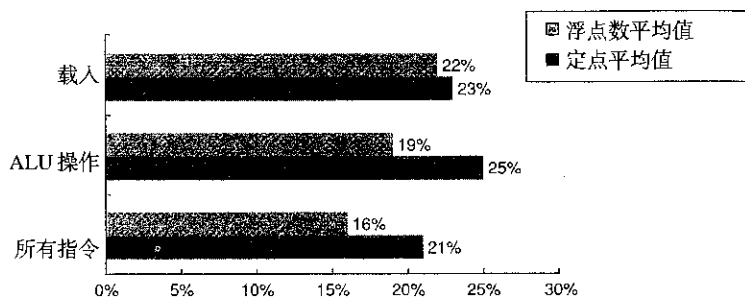


图 B.9 大约四分之一的数据传输和定点 ALU 操作有一个立即数操作数。图中最下面的横条表明在定点程序中有 1/5 的指令用到立即数，在浮点程序中这个比例为 1/6。对于载入指令，载入立即数的指令会把 16 位载入到 32 位寄存器的任意一半中。这些立即数载入并非严格意义上的载入，因为它们并不访问存储器。在某些情况下，两个被载入的立即数有可能被用来载入一个 32 位的常量，但是这种情况很少出现（对 ALU 操作来说，常数数量所引起的移位也作为含有立即数的操作）。获得这些数据所用的程序及计算机与图 B.8 的相同

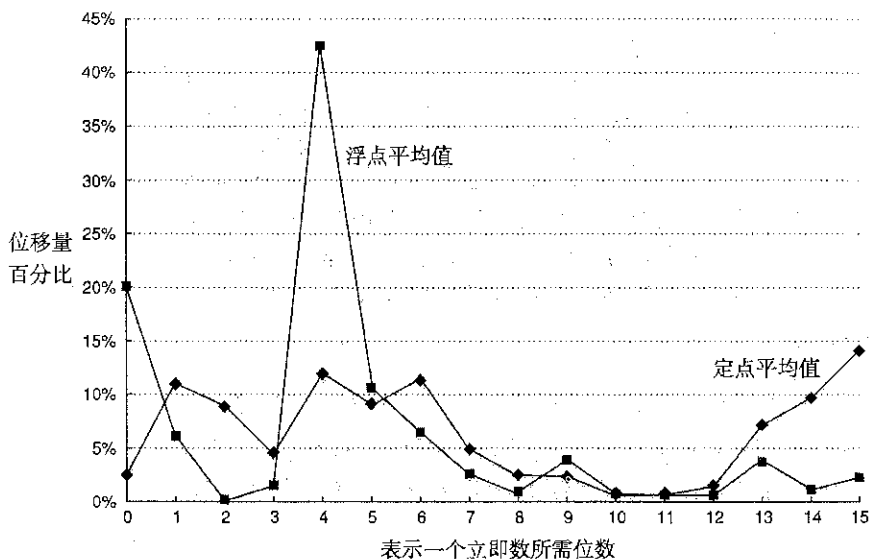


图 B.10 立即数的数值分布。 x 轴代表要表示一个立即数所需要数值的位数——0 表示立即数的数值为 0。大多数立即数的值是正的，在 CINT2000 中 20% 的立即数是负的，CFP2000 中为 30%。这些数据是在一台 Alpha 计算机上测得的，最大的立即数为 16 位（所测程序与图 B.8 相同）。在一台支持 32 位立即数的 VAX 计算机上进行相同的测试，结果显示有 20%~25% 的立即数大于 16 位。这样，16 位大约占 80%，8 位大约占 50%

B.4 操作数的大小与类型

操作数的类型是如何指定的呢？主要有两种方法：第一种，操作数类型可以通过操作码的编码来指定，这也是最常用的方法；第二种，数据上面附带一个由硬件解释的表示数据类型的符号，这个符号指定操作数的类型以及相应的操作。数据上带有类型标记的计算机现在已很难见到。

首先来分析桌面计算机和服务系统结构。一般来说，操作数的类型——例如定点、单精度浮点数和字符型等——有效地指定了操作数的大小。常见的操作数类型包括字符（1 个字节）、半字

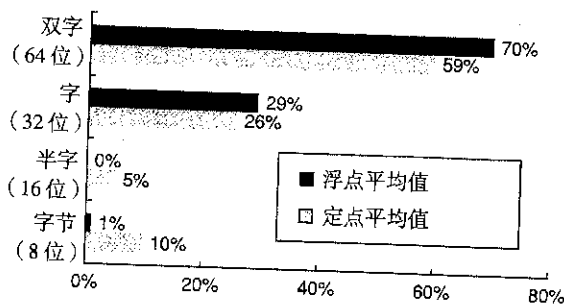
(16位)、字(32位)、单精度浮点数(也是一个字)和双精度浮点数(两个字)。定点几乎都用二进制补码表示,字符几乎都是ASCII编码格式,但是,随着计算机的国际化,16位Unicode编码(Java中使用)的使用越来越广泛。20世纪80年代初期之前,大多数计算机厂商还是各自使用自己的浮点数表示方法。之后几乎所有计算机都使用一个相同的标准来表示浮点数,即IEEE 754标准。IEEE浮点数标准的详细内容请参见附录I。

有些系统结构的计算机提供对字符的操作,然而这些操作通常很有限,只能对字符串中的每个字符分别进行处理。常见的这类操作有比较和传送。

对于商业应用,有些系统结构的计算机提供了十进制数字格式,一般称为压缩十进制或者二进制编码十进制,4个二进制位用来对0到9进行编码,每两个十进制数字存放在一个字节中。数值字符串有时称为非压缩十进制,用来在压缩和非压缩编码之间进行转换的操作称为压缩和解压缩。

使用十进制操作数的原因之一就是得到了与十进制数精确匹配的结果,因为有些十进制数小数没有等价的二进制表示。例如,一个很普通的十进制小数0.10,在二进制中需要一个无穷的位序列来表示:0.000110011...₂。所以,十进制中的精确计算在二进制中变成了不精确计算,这在金融事务中是很严重的问题(附录H中有更详细的有关精确计算机的内容)。

基准测试程序使用字节或者字符、半字(短整型)、字(整型)、双字(长整型)和浮点数据类型。图B.11显示了这些程序所访问存储器对象大小的动态分布。在决定哪些类型比较重要并且需要有效支持时,数据类型的访问频率能提供很大的依据。计算机应该支持64位访问路径,还是用两个周期来访问一个64位的双字呢?前面已经看到字节访问操作需要一个对齐网络,那么支持字节作为原始数据类型有多重要呢?在图B.11中,存储器访问用来检查被访问数据的类型。



图B.11 基准测试程序中数据访问的大小分布。由于系统为64位地址,双字数据类型可用于浮点程序中的双精度浮点数和存储器地址。在32位地址的计算机中,64位地址将被32位地址取代,定点程序中几乎所有的双字访问都转换成单字访问。

在一些系统结构中,寄存器的数据可能以字节或者半字来访问。然而,这种情况是很少发生的。在VAX计算机中,统计数据表明这种情况不超过所有寄存器访问的12%,大约占这些程序中所有操作数访问的6%。

B.5 指令系统的操作

大多数指令集系统结构都支持的操作可以按图B.12的分类方法进行分类。所有这些指令系统有一条共同的规律,就是指令系统中使用最多的是一些简单指令。例如,图B.13列出的10条简单指令在流行的Intel 80x86计算机上运行的一系列定点程序的所有指令中占96%。因此,它们是最常用的指令,执行起来应该尽量快。

操作类型	举例
算术和逻辑运算	定点算术和逻辑操作: 加、减、与、或、乘、除
数据传输	load-store 指令(在存储器寻址的计算机上是传送指令)
控制	转移、跳转、过程调用和返回、陷阱
系统	操作系统调用、虚拟存储器管理指令
浮点	浮点操作: 加、乘、除、比较
十进制	十进制加、十进制乘、十进制到字符的转换
字符串	字符串传送、字符串比较、字符串匹配
图像	像素、顶点操作、压缩/解压缩操作

图 B.12 指令操作的分类及举例。一般所有的计算机都提供三类指令。根据系统结构不同指令系统对系统功能的支持有很大的差别,但是所有的计算机都必须提供对基本系统功能的支持。后三类指令的支持情况各不相同,从没有到被扩充至很大的指令系统的情形都有。只要准备运行的应用程序有浮点操作,计算机就提供浮点指令。有时这些指令可能在一个可选的指令系统中。十进制和字符串指令有时开始就包含在指令系统中,正如 VAX 或者 IBM 360 中那样,或者由编译器通过简单的指令合成。图形指令一般对大量不太大的数据项进行并行操作,例如,对两个 64 位操作数进行 8 次 8 位的加法

如前所述,图 B.13 中的指令几乎可以在所有系统中的任何一个应用程序中找到,包括桌面程序、服务器程序及嵌入应用程序,它们也可能包括图 B.12 中的其他指令,这主要取决于指令系统所包括的数据类型。

排名	80x86 指令	定点平均值(占有所有执行指令的百分比)
1	载入	22%
2	条件转移	20%
3	比较	16%
4	存储	12%
5	加	8%
6	与	6%
7	减	5%
8	register-register 传输	4%
9	调用	1%
10	返回	1%
总计		96%

图 B.13 80x86 最常用的 10 条指令。这个表中主要是简单指令,而且它们占有所有执行指令的 96%。这些数据是 SPECint92 中 5 个程序运行结果的平均值

B.6 控制流指令

由于转移和跳转等控制流行为的测试独立于其他的测试和程序,而且与前面章节涉及的操作区别很大,因而下面介绍控制流指令的作用。

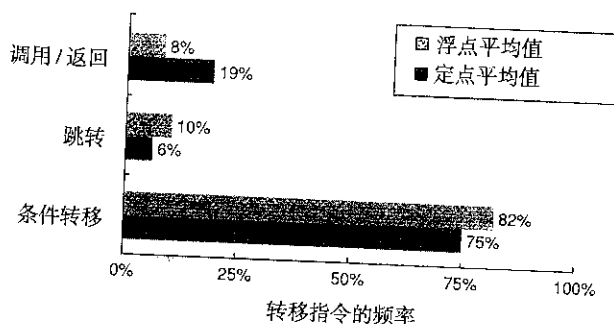
改变程序流程的指令有多种名称。在 20 世纪 50 年代一般称为**转移**,从 1960 年起开始改为分支。后来,计算机引入了新名词。本书用**跳转**来表示无条件转移,用**转移**来表示条件转移。

控制流指令可以分为 4 类:

- 条件转移。
- 跳转。

- 过程调用。
- 过程返回。

我们希望了解这些指令出现的相对频率,因为它们各不相同,可能会使用不同的指令,也可能有不同的行为。图B.14列出了在一台load-store计算机上运行基准测试程序过程中这几种控制流指令的使用频率。



图B.14 控制流指令分为三类:调用或返回、跳转和条件转移。每一类用一个长条来表示,显然条件转移占主要部分。这些数据与图B.8使用同样的计算机和同样的测试程序

控制流指令的寻址方式

控制流指令需要指明转移的目标地址。在大多数情况下,目标地址会在指令中明确地表示出来。过程返回则是例外,因为编译时不知道返回地址。最常见的方式是使用基于程序计数器(PC)的位移量来指定目标地址。这种类型的控制流指令称为PC相对寻址指令。PC相对寻址转移或者跳转的优点很明显,因为目标通常都和当前的指令离得不远,而且使用相对偏移地址可以缩减指令的长度。使用PC相对寻址还可以使程序的运行与载入的位置无关,这称为位置无关,还可以减少程序在链接时的工作量,对在执行时才链接的程序也有好处。

编译时不知道目标地址时,为了实现返回和间接跳转,还需要一种与PC相对寻址不同的实现方法。这种方法必须能够动态地指定目标地址,这样才能在运行时改变目标地址。这个动态地址可能像命名一个寄存器来存放目标地址一样简单,另外一种可能是跳转允许使用各种寻址方式来指定目标地址。

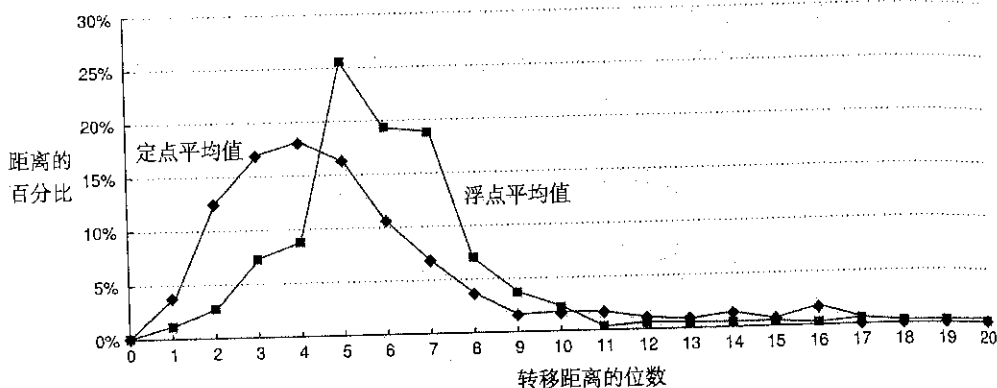
这些寄存器间接跳转还用于其他四个重要的地方:

- 许多程序设计语言中都有的分支选择语句 **case** 或者 **switch** (在多个可选项中选择一个)。
- 诸如C++, Java等面向对象语言中的**虚拟函数或方法**(根据调用参数数据类型的不同,使用不同的处理程序)。
- 诸如C, C++中的**高阶函数或函数指针**(函数可以视为参数来传递,类似面向对象的功能)。
- **动态共享库**(允许库只有当程序实际调用它时才载入存储器并链接,而不是在程序运行前静态地载入并链接)。

在上述四种情况下,编译时都不知道目标地址,因此通常在寄存器间接跳转之前,才把地址从存储器载入到寄存器中。

由于转移指令通常使用PC相对寻址来指定目标地址,因此,一个关键的问题就是跳转的目标地址离跳转指令有多远。了解跳转距离的分布有助于选择转移所使用的位移量,进一步影响到指令

的长度和编码。图B.15列出了指令中PC相对转移指令转移距离的分布情况。大约75%的转移都是前向的。



图B.15 用转移指令与目标之间的指令数来表示转移距离。在定点程序中最常见的转移间隔可用4~8位来编码。这告诉我们较短的位移量字段对转移通常是足够的,设计者采用比较短的位数来放置转移地址位移量,可以提高指令的密度。这些测量是在一台load-store结构(Alpha系统结构)的计算机上进行的,而且所有的指令在字范围内对齐。VAX系统结构对同样的程序所需的指令更少,因而转移距离会更短。同样,如果机器允许指令在任意字节范围内对齐,位移量所需的位数也会改变。测得这些统计数据的计算机和程序与图B.8的相同

条件转移的可选方案

因为大多数改变程序流程的指令是转移,所以如何指明转移条件就显得十分重要。图B.16列出了目前使用的三种方法和它们各自的优缺点。

名称	举例	如何测试条件	优点	缺点
条件码(CC)	80x86, ARM, PowerPC, SPARC, SuperH	由ALU操作设定的某些特定位,可能是由程序控制的	有时条件可由设置	CC是附加状态。条件码强制改变指令顺序,因为它把信息从一条指令传送给一个转移
条件寄存器比较并转移	Alpha, MIPS, PA-RISC, VAX	用比较结果测试任意寄存器比较是转移的一部分,通常比较只限于子集内部	简单 一个转移是一条而不是两条指令	占用一个寄存器 对流水线执行来说,一条指令要做的事情可能太多了

图B.16 评估转移条件的主要方法及其优缺点。虽然条件码也可以由其他用途所需的ALU操作设置,但是程序测试结果表明这种方法很少使用。条件码实现的主要问题是条件码有很多种指令,或是由众多指令中随便选取的一部分来设定,而不是由指令中的一个位来控制的。拥有比较及转移指令的计算机通常限制比较的集合,并且使用条件寄存器进行更复杂的比较。通常基于浮点数比较的转移和基于定点比较的转移的实现技术不相同。由于基于浮点数比较的转移次数比基于比较的转移次数少很多,所以这样做很合理

转移最值得注意的特点就是大量的比较仅仅是简单的测试,而且其中大部分是测试某个值是否为零。因此,一些系统结构把这些情况作为特殊情况处理,特别是使用比较指令和转移指令时。图B.17列出了条件转移中使用的不同比较情形的频率分布。

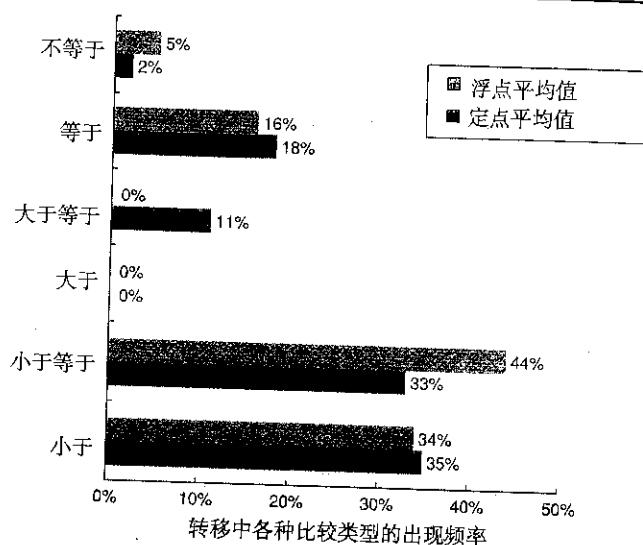


图 B.17 不同类型条件转移中的频率。小于（或等于）转移在这种编译器和系统结构的组合中占据主要地位。这包括转移中定点的比较和浮点的比较。用来进行测量的计算机和程序与图 B.8 中的相同

过程调用的可选方案

过程调用和返回包括控制转移，可能还包括状态的保存，至少返回地址是保存在某个地方，可以是专门的链接寄存器，也可以是通用寄存器。有一些旧的系统结构提供了保存多个寄存器的机制，新的系统结构需要编译器生成指令来保存或恢复寄存器。

有两种基本、传统的方法用来保存寄存器：调用者保存和被调用者保存。调用者保存意味着调用者在调用其他过程时，必须保存在调用过程之后还要使用的寄存器，被调用者则无须维护这些寄存器。被调用者保存意味着被调用的过程必须保存它要使用的寄存器，调用者则不受这种限制。有时候，由于两个不同的过程都用到全局变量的访问模式，因此必须使用调用者保存方法。例如，假设进程 P1 调用另一个进程 P2，而且这两个进程对同一个全局变量 x 进行操作。如果 P1 在调用 P2 之前已经将 x 载入到寄存器中，那么必须在调用 P2 之前把 x 存在一个 P2 可以访问的地方。编译器很难发现被调用的进程使用了哪些变量，因为每个部分是分别编译的。另外还有一种情况，被调用的进程 P2 可能本身并不访问 x ，而是调用了另外一个进程 P3 来访问变量 x 。因为这种情况很复杂，在大多数编译器中，由调用的进程保存本次调用中可能用到的所有变量。

在两种传统方法都可以使用的情况下，有的程序适用于调用者保存，有的程序适用于被调用者保存。因此，大多数实际使用的编译器会结合这两种方法。这种方法需要应用程序二进制接口 (ABI) 来指定变量采用哪种保存方式。本附录的后面部分将讨论自动保存寄存器的复杂指令与寄存器需求之间的矛盾。

总结：控制流指令

控制流指令是使用得最多的一类指令。虽然条件分支指令已经有多种选择，但最好还是能够向前或者向后跳转几百条指令，即 PC 相对分支位移量应该至少有 8 位。我们也希望无条件跳转指令的寄存器间接寻址方式和 PC 相对寻址方式也能够支持返回，像目前的许多计算机系统支持的其他许多新特性一样。

至此,我们已经从汇编程序员和编译器开发人员的角度了解了指令集系统结构。结合load-store系统结构讨论了位移量寻址、立即寻址和寄存器间接寻址方式。这些数据是8, 16, 32, 64位定点和32, 64位浮点数据。所讨论的指令包括简单指令,用来调用过程的PC相对寻址条件分支、跳转和链接指令,以及用于过程返回(还有一些其他用法)的寄存器间接跳转。

下面来研究如何用一种易于硬件实现的形式表示这种系统结构。

B.7 指令系统的编码

显然,上面讨论的问题将会直接影响到指令如何编码成处理器可以执行的二进制代码形式。这种形式不仅会影响到编译后程序的大小,还会影响到指令在处理器中的实现,因为处理器必须对这个指令译码,并尽快确定指令中的操作和操作数。操作一般由操作码的字段来确定。正如下文要讲到的,最重要的是如何把寻址方式和操作通过编码结合到一起。

这种结合取决于指令系统都包含哪些寻址方式以及操作码与寻址方式的相关性。一些计算机有1~5个操作数,每个操作数有10种可能的寻址方式(见图B.6)。对这么多的信息来说,一般每个操作数还需要一个独立的寻址标识符来指定使用哪种寻址方式来访问操作数。另外一个极端的情况是只有一个存储器操作数、一到两种寻址方式的load-store系统结构的计算机。显然,在这种情况下,寻址方式可以作为操作数的一部分进行编码。

对指令进行编码时,寄存器和寻址方式的数量对指令的大小有明显的影响,因为在一条指令中,寄存器字段和地址字段都可能出现很多次。实际上,大多数指令用于寻址方式和寄存器编码上的位数比用在操作码上的位数还要多。在对指令进行编码时,应该在下面的几个因素之间找到一个最佳的平衡点:

1. 希望有尽可能多的寄存器和寻址方式。
2. 平均指令长度中用来存放寄存器和寻址方式的字段尽量少,从而使程序尽量小。
3. 指令的长度应该易于用流水线处理(指令应易于译码的重要性在第2章和附录A有详细介绍)。首先,计算机希望所有的指令长度应该是字节的整数倍,而不是随意的位数。许多桌面和服务器的为了使用固定的指令长度,不得不在指令的平均数长度上进行折中。

图B.18所示为三种常见的编码方式。第一种是变长编码,它允许所有的操作使用所有的寻址方式。这种方法适合寻址方式和操作比较多情形。第二种是定长编码,它把操作和寻址方式组合在操作码里,通常所有的指令长度都相同。这种方式适合于寻址方式和操作比较少情况。上述两种方式之间的取舍实质上是在程序大小和处理器译码难易程度之间的取舍。变长编码总是试图用最少的位数来表示程序,但是每条指令在指令长度及其可完成的操作上有很大的差别。

下面来看一个变长编码的例子,这是80x86的一条指令:

```
add EAX, 1000 (EBX)
```

add是一条有两个操作数的32位加法指令,操作码占1个字节,一个80x86地址标识符占1或2个字节,用来确定源或目标寄存器(EAX)和寻址方式(本例中是位移量寻址)及第二个操作数的基址寄存器(EBX)。本例中用一个字节指定操作数。在32位模式下(见附录J),地址字段占1到4个字节。1000超过了 2^8 ,所以本条指令的长度为 $1 + 1 + 4 = 6$ 字节。

80x86指令的长度从1字节到17字节不等,它的程序通常比使用定长编码(见附录J)的RISC系统结构的程序要小。

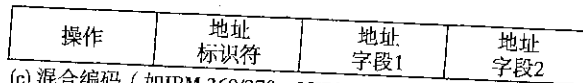
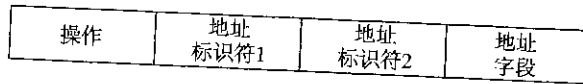
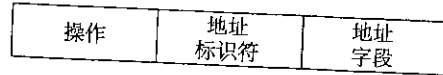
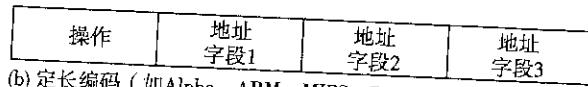
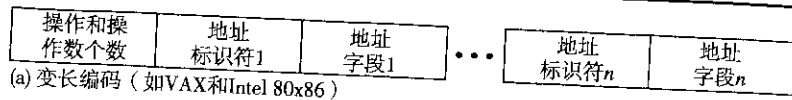


图 B.18 三种常见的指令编码方法：变长编码、定长编码和混合编码。变长编码可以支持很多操作数，地址标识符指定每个操作数的寻址方式和操作数标识符的长度。由于不包括未用的字段，这种方式产生的代码通常是最小的。定长编码的操作数个数相同，寻址方式由操作码的一部分指定（见附录 C 中的图 C.3）。这种方式产生的程序代码通常是最大的。虽然这些字段的位置是基本固定的，但是在不同指令中有不同的用途。综合这两种方法的优缺点，混合编码可以使用几种固定的编码格式，增加一两个用来指定寻址方式的字段和一两个用来指定操作数地址的字段（见附录 D 的图 D.7）

了解了这两种方法之后，下面介绍第三种方法，这也是一种混合方法：减少过多的指令长度类型，以减轻多种结构的指令带来的工作负担，但仍提供多种指令长度以减少代码长度。以下将给出实例。

RISC 中的精简代码

随着 RISC 计算机在嵌入式领域的应用，由于成本和代码量大小的的重要性，32 位定长编码方式有些力不从心。为了适应这种情况，许多厂家提出了他们自己的混合编码方式，包括 16 位和 32 位指令。较短的指令支持较少的操作，较小而且相近的地址和较少的寄存器，同时还放弃了典型 RISC 的三地址指令格式而采用两地址格式。附录 J 有两个例子：ARM Thumb 和 MIPS MIPS16，二者都声称可以将代码减缩 40%。

与上面的指令系统扩展相反，IBM 将其标准指令系统压缩，然后添加新硬件来译码（如果指令 Cache 未命中，则要从存储器中读取时进行译码）。这种指令 Cache 只保存 32 位指令，压缩的指令留在存储器、ROM 和磁盘中。MIPS16 和 Thumb 的优点是指令 Cache 工作起来似乎增大了 25%，而 IBM 的 CodePack 意味着编译器无须修改就可以处理不同的指令系统，而且指令的译码也很容易。

CodePack 随着 PowerPC 上程序的执行而开始进行编码压缩，然后，把压缩的结果表载入到芯片上的一个 2 KB 大小的表中，因此，每个程序都有自己单独的编码。为了处理条件分支（通常不在字的对齐范围内），PowerPC 在存储器中建立一个 Hash 表并且将压缩和未压缩的地址进行映射，与 TLB（变换旁视缓冲器，参见第 5 章）类似。这张表通过缓存最常用的地址来减少存储器的访问次数。IBM 宣称这种方法减少了 10% 的总体性能，但可以将代码压缩 35%~40%。

Hitachi 公司为嵌入式应用（见附录 J）开发了一种简单的 16 位定长指令系统——SuperH。为了配合较短的指令格式和较少的操作，这种指令系统只有 16 个寄存器而不是 32 个，但其他方面均与典型的 RISC 系统类似。

总结：指令系统编码

前几节讨论了指令系统设计的一些问题，即指令系统采用什么样的编码方式。如果关注生成代码量的大小，就应该选择变长编码方法；如果更关注程序的运行性能，就应该选择定长编码方法。附录 D 中给出了对系统结构进行选择的 13 个例子。第 2 章和附录 A 会进一步讨论处理器的灵活性和性能之间的相互影响。

至此，有关 MIPS 指令集系统结构的背景知识已经介绍完毕，但我们还需要了解编译技术及其对程序的影响。

B.8 相关问题：编译器的角色

如今，几乎所有的桌面程序和服务器程序都是用高级语言编写的。在这种情况下，由于运行的大多数指令都是编译器的输出，因此一个指令集系统结构本质上就是编译器的最终目标。早期针对这些应用，指令集系统结构方面的设计通常主要考虑如何使汇编语言程序或特定程序内核设计更容易。因为编译器对计算机的性能影响很大，如果想要设计或高效率地实现一个指令系统，就要理解今天的编译技术。

之前，通常把编译技术及其对计算机硬件性能的影响和系统结构及其性能分隔开来，这就好比经常把系统结构与它的实现分开一样。这种分隔在今天的编译器与机器之间几乎是不可能的。系统结构的选择除了会影响机器产生代码的质量外，还将影响为该机器构建一个高效编译器的复杂度。

本节将主要从编译器的角度来讨论指令系统的一些重要目标。从回顾一个现代编译器开始，然后讨论编译器技术如何影响系统结构方面的设计，以及系统设计者如何才能使其编译生成高质量的代码，最后介绍编译器和多媒体应用，而多媒体应用正是一个关于编译器开发者和系统设计者合作的反例。

近来编译器的结构

首先看一下当前最理想的编译器应满足什么样的条件。图 B.19 所示为近期编译器的结构。

编译器开发者的首要目标就是正确性——所有有效的程序都应该能够被正确地编译。另一个目标是编译出来的代码的执行速度。通常，其他的目标都应该以这两个目标为前提，它们包括：快速编译、调试支持以及语言之间的互操作性。一般而言，编译器把高级的、更加抽象的表示方式逐步转换成相对低级的表示方式，最终到达指令系统。这种结构有助于管理复杂的转换，同时也易于编写出一个正确的编译器。

开发一个高效编译器的复杂度主要受优化程度的限制。虽然多遍编译结构降低了编译器的复杂度，但同时也意味着编译器必须有序执行各种转换。在图 B.19 的优化编译器的图表中，可以发现某些高级优化在得到结果码之前就已经开始运行。一旦做了这样的转换，那么编译器将不可能重新返回并再次扫描所有步骤，则需取消所进行的转换。从编译时间和复杂度两方面考虑，都是不能容忍的。因此，编译器假定后面的步骤有能力处理某些问题。例如，编译器一般在得知被调用过程的确切大小之前挑选出要展开的过程。编译器开发者把这个问题称为状态-时序问题。

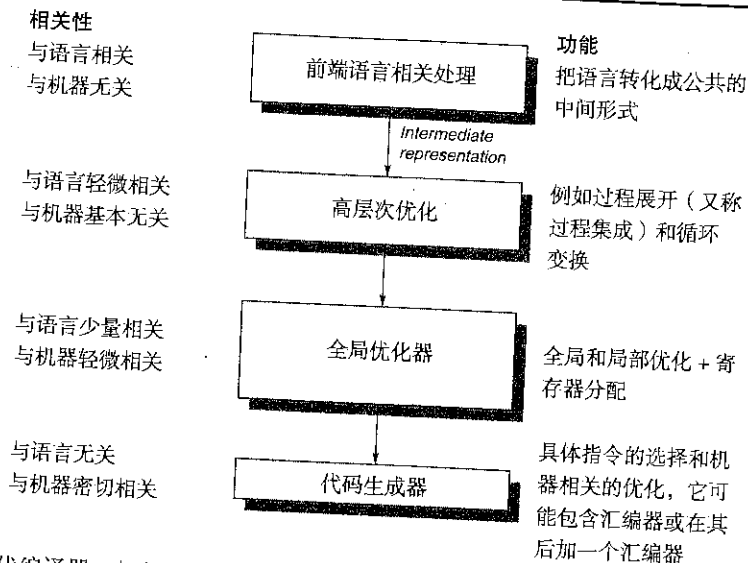


图 B.19 近代编译器一般包括 2~4 遍扫描，而性能更高的编译器包括更多次的扫描。这种结构最大限度地提高了在相同输入前提下以不同的优化级别编译的程序可以产生相同输出的可能性。当编译的目标是更高的编译速度同时较低的代码质量能被接受时，优化扫描可以被跳过。一遍扫描简单来说就是编译器读取并且翻译整个程序的一个步骤（这里步骤这个词经常可以和遍数这个词互换）。优化扫描被设计成一个可选项。由于优化扫描是独立的，因此不同的语言可以使用相同的优化与代码生成步骤。对于一种新的语言，只需要一个新的前端程序就可以了。

这种转换顺序是怎样与指令集系统结构相互作用的呢？这里有一个在优化时使用的不错的例子，称为**全局公共子表达式消去法**。这种优化找出计算同一个变量表达式的两个实例，并把第一次计算出的值存入一个临时变量中。此后它利用这个临时变量中的值，而省略表达式的第二次计算。

为了使这种优化有效，这个临时变量必须存放于寄存器中。否则，如果将这个临时变量存放在存储器中，此后，重新载入所需的开销就会抵消因为没有重复计算表达式而得到的优化。因此，实际上存在这样一种情况：有时这种优化反而会使代码的执行速度变慢，因为优化使用的临时变量未被保存在寄存器中。时序使得这个问题变得更加复杂，因为寄存器分配一般是在全局优化扫描快要结束时进行的，发生在代码生成之前。因此，执行这种优化的优化器须假设寄存器分配器将为临时变量分配寄存器。

现代编译器的优化方式可以依据转换风格的不同而分为以下几类：

- 高层优化一般在源码上进行，同时把输出传递给以后的优化扫描步骤。
- 局部优化仅在一系列代码片段之内（编译器设计者把它称为基本段）将代码优化。
- 全局优化将局部优化扩展为跨越分支，并且引入一组针对优化循环的转换。
- 寄存器分配将寄存器和操作数联系起来。
- 处理器相关的优化总是充分利用特定的系统结构。

寄存器分配

由于寄存器分配在加速代码和使其他优化发挥作用这两个方面扮演着一个核心角色，因而它是最重要的优化方案之一。当今的寄存器分配算法都基于一种称为**图着色**的技术。图着色的基本思想是构造一个图，用它来代表分配寄存器的各个可能候选方案，然后用此图来分配寄存器。即如何用

有限的颜色使图中相邻的节点分别着以不同的颜色。这种方式所要强调的是要100%地完成活动变量的分配。图着色问题通常是一个图大小的指数函数（NP完全问题），不过有些启发式算法在实际中效果不错，可产生近于线性时间运行的分配。

在全局分配中如果能够有16个（最好更多）通用寄存器用于整型变量，同时另有其他的寄存器用于浮点数，在这种情况下，图着色将会正常工作。遗憾的是图着色在寄存器数目比较少的情況下并不能够很好地工作，这是由于图着色的启发式算法在这种情况下有可能会失效。

优化对性能的影响

有时候，把某些简单优化（如局部优化和机器相关的优化）从代码生成器的转化中实现分离是很困难的。图B.20中给出了一个典型优化例子。图B.20中的最后一列给出了所列优化转换用于源程序的频率。

优化名称	说明	在优化转换中所占比例
高级语言层	接近或者在源代码级，与机器无关	未被测试
过程集成	用过程体替代过程调用	未被测试
局部	在一系列代码中	
公共子表达式消去	用一份副本替代同一个计算的两个实例	18%
常量传递	把所有分配了一个常量的变量的实例用该常量替换	22%
堆栈高度缩减	重新安排表达式树使表达式计算所需要的资源最少	未被测试
全局	跨越分支	
全局公共子表达式消去	与局部的方法相同，但是这里会跨越分支	13%
副本传递	用X(如A=X)替换已经赋值以X的变量A的所有实例	11%
代码移动	把循环中每次迭代计算同一变量的代码从循环中移去	16%
归纳变量消去	在循环中简化或者消去数组地址的计算	2%
处理器相关	依赖处理器结构	未被测试
长度缩减	有很多例子，如用一个常数的加法和移位来代替乘法	未被测试
流水线调度	重新对指令进行排序以改进流水线的性能	未被测试
分支偏移优化	选择到达目标的最短分支位移路径	未被测试

图B.20 优化的主要类型及举例。这些数据给出了不同优化方法的相对使用频率。第3列所列出的是12个小型FORTRAN和Pascal程序中一些普通优化的静态频率。在这次测试中，有9个由编译器进行的局部和全局优化。图中给出了6种优化方式的数据，剩下的三种方式在静态优化中大约占18%。“未被测试”表示相应类型的优化未测试。依赖机器的优化经常在代码生成器中进行，在本次实验中没有测试。数据来自[Chow 1983]（数据收集使用Stanford UC DODE编译器进行）

图B.21所示为两个程序采用不同的优化方法对指令执行时间产生的影响。优化后的程序大约比未优化的程序少执行25%~90%的指令。该图说明在为指令系统新增加功能之前，应先查看优化代码的重要性，因为编译器可能会把设计者要改进的指令去掉。

编译技术对系统结构设计者的决策所产生的影响

编译器与高级语言之间的相互作用对程序如何使用指令系统结构有显著的影响。这里有两个重要的问题：变量是如何分配和寻址的？需要多少个寄存器才能有效地分配变量？为了解决这些问题，必须首先了解当前语言在何处分派数据：

- **堆栈**被用来分配局部变量。在过程调用或者过程返回时堆栈会相应地增大或缩小。堆栈中的对象是相对于堆栈指针来寻址的，并且多半是标量（单变量）而不是数组。堆栈被用于激活记录，而不是用来计算表达式。因此数值一般不会被压入或弹出堆栈。
- **全局数据区**用来分配静态声明的对象，如全局变量和常量。这些对象中大部分是数组或者是其他聚合数据结构。
- **堆**用来分配那些不适于放在堆栈中的动态的对象。堆中的对象都要用指针来访问，一般不是标量。

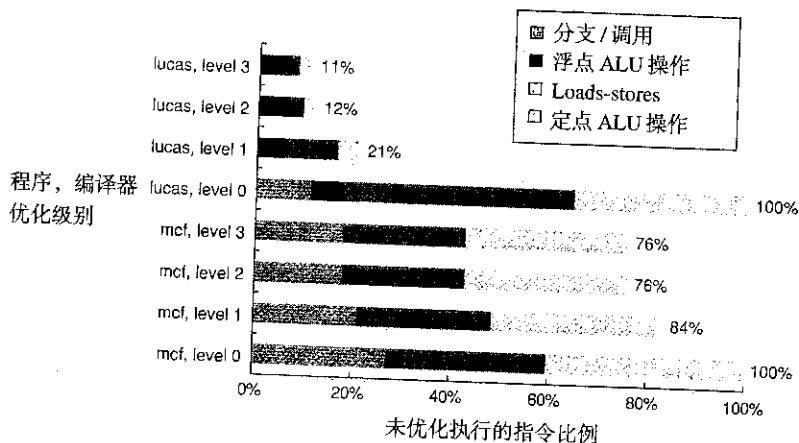


图 B.21 当编译优化级别不同时，SPEC2000 中的 lucas 和 mcf 两个程序指令条数的变化。0 级等同于未优化代码。1 级包括局部优化、代码调度和局部寄存器分配。2 级包括全局优化、循环转换（软件流水线），以及全局寄存器分配。3 级加了过程集成。这些实验是在 Alpha 的编译器上运行的

为堆栈分配对象进行寄存器分配要比对全局变量更加有效，而寄存器分配对于堆分配对象来说基本上是不可能的，因为后者是使用指针进行访问的。全局变量以及某些堆变量也不可能进行分配，因为它们具有“别名”，即有多种不同方式可以访问这个变量的地址，这使得将它放入寄存器是非法的（在当今的编译器技术中大多数堆变量可有效地使用别名）。

例如，请看下面的代码序列，其中 & 返回一个变量地址，而 * 则是取地址的值：

```

p = &a      -- 取 a 的地址给 p
a = ...     -- 直接对 a 赋值
*p = ...    -- 使用 p 对 a 赋值
...a...     -- 访问 a

```

通过对 *p 赋值将变量 a 分配到寄存器中是不可能产生正确代码的。别名会带来很多问题，原因是决定某一指针可能指向哪些对象是非常困难甚至是不可能的，因此编译器必须非常保守。许多编译器在某一过程中只要有一个指针有可能指向该过程的任何一个局部变量，就不允许把此过程中的任何局部变量分配到寄存器中。

系统结构设计者对编译器设计者的技术支持

当今，编译器的复杂性并不是来源于一些简单语句，如 $A = B + C$ 。大多数程序是局部简单的，并且简单的转换就能够很好地处理它们。确切地讲，复杂性是由于程序的庞大及其复杂的全局交互作用造成的。这意味着编译器必须决定哪一种代码顺序最好，而且一次只能做一步。

编译器设计者工作时通常会遵循一个原则：确保经常出现的事件要尽量快，而不经常出现的事件则一定正确。即如果知道哪些事件经常发生，哪些时间很少出现，而且为这两种事件生成代码又都很简单，那么很少发生的事件所生成代码的质量就并不十分重要了，但一定要正确！

一些指令系统特性有助于编译器设计者。这些特性并不是不可违反的规则，它们只是一些指导性的准则，会使编译器更容易编写，且生成正确而且有效的代码。

- **规则性：**要达到这一点，则指令系统中的三种主要元素——操作、数据类型和寻址方式必须是正交的。如果一种系统结构中的两种特性相互独立，则称之为“正交”。以操作和寻址方式为例，如果每一种操作可采用任意一种寻址方式，则称此两者为正交。这有助于简化代码的生成，并且当所要生成的代码在编译器的两遍扫描中分开时会变得特别重要。该性质的一个反例是：对某一类指令，限制其可用的寄存器。专用寄存器结构的编译器在这种情况下会不知所措：它会发现系统中有很多寄存器，惟独没有要用到的那一种。
- **提供原语，而不是解决方案：**只适合某一种语言结构的特性通常没有太大用处。企图去支持高级语言只可能对一种语言可行，否则就是与该语言的正确且有效的实现有偏差。B.10节给出了一些例子，这些例子说明了这些尝试是如何失败的。
- **在取舍时考虑简化的折中：**对编译器的设计者们来说，最棘手的工作之一就是对每段代码都要计算出哪一个指令序列是取优的。在早些时候，指令条数或者全部代码的大小可能是很好的度量指标，但是，在有了Cache和流水线之后，折中变得相当复杂。设计者所做的任何一件事情，只要能够帮助编译器设计者了解两种互相可以替换的代码序列的代价，就会有助于改进代码。在折中较复杂的情况下，最典型的例子之一就是在寄存器-存储器系统结构上决定一个变量应该被访问多少次后才将它取到寄存器比较好。这条界限很难划分，实际上，即使在相同系统结构的不同模块中，它可能也会不同。
- **对于编译时作为常量的数值量，提供能将其确定为常量的指令：**编译器设计者不喜欢机器在执行时还要去解释一个编译时已知的值。这个原则的一个反例就是指令对一个在编译时已确定的值进行解释。例如，VAX过程调用指令动态地解释掩码，以确定在调用中需要保存哪些寄存器，但这个掩码在编译时已经确定了（见B.10节）。

编译器对多媒体指令的支持

对于在一个时钟周期内处理数条较小数据项的单指令流多数据流指令（SIMD）来说，设计者会忽略前一小节中的内容。这些指令是解决方案而不是原语，它们的寄存器数量少，数据类型与现在的高级语言不匹配。系统设计者希望能找到一种代价较小的方法以对用户有所帮助，但事实上，只有很少的底层图形库使用它们。

单指令流多数据流指令事实上可以看做是一个包含编译技术的成功系统结构的压缩版本。向量系统结构（在附录G中介绍）处理向量数据。多媒体内核最初是为科学计算而设计的，但也常被向量化。所以，可以把Intel MMX和PowerPC AltiVec看做是较小的向量机：MMX的向量是8个8位、4个16位或2个32位的元素；AltiVec的向量长度是MMX的两倍。它们共同的特点是寄存器较大，元素项较小且相互邻近。

这种压缩的结构确定了向量寄存器的大小：MMX元素长度总和不超过64位，AltiVec不超过128位。为了扩展到128位，Intel增加了一组指令：流式SIMD扩展（SSE）。

向量计算机的另一个特点是通过一次载入多个数据而减少存储器访问时延，并且使数据传输与指令执行重叠。向量寻址方式的目标就是收集存储器中分散的数据，并组织成紧凑的形式，以便快捷地操作并将结果存回目的地。

经过多年的发展,传统的向量机增加了跨越式寻址和聚集/分散寻址两种寻址方式,以增加向量化程序。跨越式寻址在相邻的访问之间跳过固定字数,因此,顺序寻址也称为单元跨越寻址。聚集/分散寻址从另一个向量寄存器中得到它们的地址,与寄存器间接寻址类似。从向量机的角度看,这些短向量SIMD向量机仅支持单元跨越式寻址,每次存储器存储或载入只能操作一个存储器单元。由于多媒体数据大多是存储器中连续的流式数据,跨越式寻址和聚集/分散寻址可以很容易地实现向量化。

当然,与这种系统结构相关的短向量也有缺点,由于其寄存器较少,寻址方式简单,它很难应用向量化的编译技术。另外就是迄今尚无编程语言支持这么短的数据项的操作。因此,这种SIMD指令只能在手工代码库中用到。

例题 像素的颜色有两种表示:RGB(红绿蓝)和YUV(发光度、色度),每个像素需3个字节表示。下面通过将RGB转化为YUV来比较向量机与MMX。转化只需要3行C代码:

```
Y = (9798*R + 19235*G + 3736*B)/ 32768;
U = (-4784*R - 9437*G + 4221*B)/ 32768 + 128;
V = (20218*R - 16941*G - 3277*B)/ 32768 + 128;
```

64位向量计算机可以同时处理8个像素。使用跨越式寻址的媒体向量计算机进行以下步骤:

- 3个向量载入(获取RGB)
- 3个向量乘法(转化R)
- 6个向量乘加(转化G和B)
- 3个向量移位(除以32768)
- 2个向量加法(加128)
- 3个向量存储(存储YUV)

用上面的C代码对8个像素进行转化需20条指令执行这20个操作[Kozyrakis2000](由于一个向量有32个64位元素,这段代码实际转化 32×8 即256个像素)。

相比之下,Intel的Web站点公布,进行同样的8个像素的转化需使用116条MMX指令以及6条80x86指令[Intel 2001]。指令数几乎是向量机的6倍,之所以会这样,是因为MMX中没有跨越式寻址,需要不断地载入并解包RGB像素,同时打包存储YUV像素。

总结:编译器的角色

这一节提出了几个建议。首先,我们希望一个新的指令集系统结构有至少16个以上的通用寄存器——不包括单独的浮点数寄存器,用图着色法来简化寄存器的分配。其次是正交,所有支持的寻址方式都可以应用于任何传输数据的指令。最后是上一小节的最后三条建议:提供原语而不是解决方案,在取舍时考虑简化的折中,不在运行时绑定常量——所有这些建议都以简化为最终目标。理解这些内容有助于指令系统的设计,扩展SIMD指令系统更重要的意义在于其市场价值,而不仅仅由于它是软硬件综合设计的成功范例。

B.9 综合: MIPS 系统结构

这一节将介绍一种称为MIPS的简单64位load-store系统结构。MIPS和RISC系列的指令集系统结构基于前几节的内容(K.3节将讨论这些系统结构怎样以及为什么会广泛使用)。首先回顾一下每一节的内容:

- B.2 节：使用 load-store 系统结构的通用寄存器。
- B.3 节：支持以下寻址方式：位移量（地址位移量有 12~16 位）、立即数（8~16 位）以及寄存器间接寻址。
- B.4 节：支持以下数据长度与类型：8 位、16 位、32 位定点、64 位定点以及 64 位 IEEE 754 浮点数。
- B.5 节：支持以下简单指令，因为它们占执行指令的绝大部分：load, store, add, subtract, move register-register 和 shift。
- B.6 节：比较指令：compare equal, compare not equal, compare less, branch（分支，有一个至少 8 位的 PC 相对地址），jump, call 和 return。
- B.7 节：如果关心性能就使用固定长度指令编码，如果关心代码大小就使用可变长度指令编码。
- B.8 节：提供至少 16 个通用寄存器以及单独的浮点数寄存器，确保所有寻址方式都可用于所有的数据传输指令，使得指令系统规模最小。这部分不包括浮点数程序，通常设置独立的浮点数寄存器。增加浮点寄存器是为了不在指令格式以及通用寄存器的速度上产生问题。这种选择是非正交的。

我们将以如何遵循上述的建议为线索来介绍 MIPS。与大多数当今的机器类似，MIPS 强调

- 简单的 load-store 指令系统。
- 设计上重视流水线（在附录 A 中讨论）效率，包括固定长度指令编码。
- 使编译器更容易产生高效的目标代码。

MIPS 是一种适合于学习和研究的系统结构模型，它是一种应用广泛（见第 1 章）且容易理解的系统结构。附录 A、第 2 章与第 3 章中会再次使用这种系统结构模型，它将是习题和编程项目的主要基础。

自从第一个 MIPS 处理器在 1985 年诞生以来，这种系统结构已经有了许多版本（见附录 J）。我们将使用它的一种称为 MIPS64 的子集，它是 MIPS 的简化版本，完整的 MIPS 指令集参见附录 J。

MIPS 的寄存器

MIPS 有 32 个 64 位通用寄存器（GPR），名称为 R0, R1, ..., R31，有时也叫定点寄存器。另外还有 32 个浮点数寄存器（FPR），名称为 F0, F1, ..., F31。它们既可以作为 32 个 32 位单精度寄存器来使用，也可以作为 32 个 64 位双精度浮点寄存器来使用（存储单精度浮点数时，FPR 的另一半未使用）。MIPS 提供单精度和双精度操作（32 位和 64 位），还提供一个 64 位寄存器上的两个单精度数的操作。

R0 的值永远是 0。我们将在后面看到怎样通过这个寄存器用简单指令系统来合成一组有用的操作。

一些特殊寄存器用来与定点寄存器交换数据。浮点状态寄存器用来保存有关浮点操作结果的数据。还有一些指令用来在 FPR 和 GPR 之间传送数据。

MIPS 的数据类型

定点数据类型有 8 位字节、16 位半字、32 位字和 64 位双字，浮点数有 32 位单精度和 64 位双精度浮点数。半字类型被加入该最小数据类型集合是因为它出现在类似 C 的语言中，在操作系统等程序中半字也很流行，因为这些程序很重视数据结构的大小。如果 Unicode 被广泛应用，这些数据

类型会更流行。单精度浮点操作数也因同样的理由被加入到MIPS数据类型集合中(请记住前文的提醒:在设计一种指令系统之前必须测试尽可能多的程序)。

MIPS64的操作是面向64位定点以及32位或64位浮点数的。字节、半字或者字在调入64位寄存器中时,用零或者符号位来填充64位寄存器的剩余部分。一旦被调入以后,它们将按照64位定点的方式进行计算。

MIPS数据传输的寻址方式

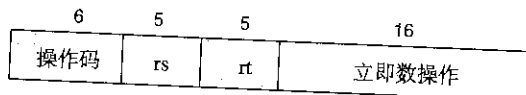
MIPS数据寻址方式只有立即数和位移量方式两种,这两种方式都是16位的。寄存器间接寻址是通过把0放入16位位移字段中完成的。16位绝对寻址字段是通过R0作为基址寄存器完成的。这样我们就得到了4种有效的模式,尽管系统结构只支持两种。

MIPS的存储器是用64位地址字节寻址的。有一个供软件选择的模式位来决定是高位字节先传格式还是低位字节先传格式。因为它是load-store系统结构,所有的存储器访问都必须通过存储器和GPR或存储器和FPR之间的载入或存储操作完成。由于支持上面提到的所有数据类型,所以与GPR有关的存储器访问可以是一个字节、一个半字或一个字。FPR可以载入或存储单精度或双精度字。所有存储器访问必须是对齐的。

MIPS指令格式

由于MIPS只有两种寻址方式,所以它们可以编码到操作码中。为了使机器更容易进行流水线操作和译码,所有指令都是32位的,其中6位是基本操作码。指令的格式如图B.22所示。这些指令格式很简单,同时还为位移量寻址、立即数或PC相对分支地址提供了16位字段。

I型指令



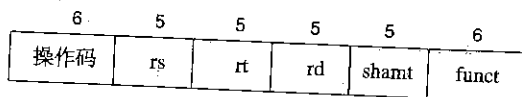
编码:加载/存储字节、半字、字、双字。

所有立即数($rt \leftarrow rs$ 立即数操作)

条件分支指令(rs表示寄存器,rd表示未使用)

跳转寄存器、跳转并链接寄存器($rd = 0$, rs表示目标,立即数为0)

R型指令

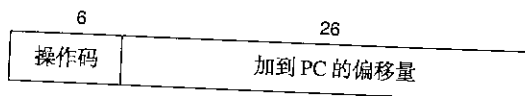


寄存器-寄存器ALU操作: $rd \leftarrow rs \text{ funct } rt$

函数编码数据通路操作: Add, Sub, ...

读/写专用寄存器和数据移动

J型指令



跳转,跳转并链接

陷阱和从异常中返回

图B.22 MIPS指令格式。所有指令都按这3种类型之一来编码,通用字段在每种格式中的位置都相同。

附录J中是另外一种MIPS系统——MIPS16,它使用16位和32位指令以提高嵌入式应用指令密度。我们会着重讨论典型的32位格式。

MIPS 操作

MIPS支持上面提到的一些简单操作,还有一些其他操作。指令大致可以分为四类:载入和存储、ALU操作、分支与跳转以及浮点操作。

所有通用寄存器与浮点数寄存器都可以被载入或存储,唯一的例外是载入R0无效。图B.23给出了载入和存储指令的例子。单精度浮点数占用一半浮点数寄存器。单精度与双精度之间的转换必须显式地进行。浮点数据格式是IEEE 754(见附录I)。图B.26列出了MIPS的所有指令。

指令举例	指令名称	含义
LD R1, 30(R2)	载入双字	$\text{Regs}[R1] \leftarrow_{64} \text{Mem}[30 + \text{Regs}[R2]]$
LD R1, 1000(R0)	载入双字	$\text{Regs}[R1] \leftarrow_{64} \text{Mem}[100 + 0]$
LW R1, 60(R2)	载入字	$\text{Regs}[R1] \leftarrow_{64} (\text{Mem}[60 + \text{Regs}[R2]])_{0}^{32} \text{ ## Mem}[60 + \text{Regs}[R2]]$
LB R1, 40(R3)	载入字节	$\text{Regs}[R1] \leftarrow_{64} (\text{Mem}[40 + \text{Regs}[R2]])_{0}^{56} \text{ ## Mem}[40 + \text{Regs}[R3]]$
LBU R1, 40(R3)	载入无符号字节	$\text{Regs}[R1] \leftarrow_{64} 0^{56} \text{ ## Mem}[40 + \text{Regs}[R3]]$
LH R1, 40(R3)	载入半字	$\text{Regs}[R1] \leftarrow_{64} (\text{Mem}[40 + \text{Regs}[R2]])_{0}^{48} \text{ ## Mem}[40 + \text{Regs}[R3]] \text{ ## Mem}[41 + \text{Regs}[R3]]$
LS F0, 50(R3)	载入单精度浮点数	$\text{Regs}[F0] \leftarrow_{64} \text{Mem}[50 + \text{Regs}[R3]] \text{ ## } 0^{32}$
LD F0, 50(R2)	载入双精度浮点数	$\text{Regs}[F0] \leftarrow_{64} \text{Mem}[50 + \text{Regs}[R2]]$
SD (R3), 500(R4)	存储双字	$\text{Mem}[500 + \text{Regs}[R4]] \leftarrow_{64} \text{Regs}[R3]$
SW R3, 500(R4)	存储字	$\text{Mem}[500 + \text{Regs}[R3]] \leftarrow_{32} \text{Regs}[R3]_{32..63}$
SS F0, 40(R3)	存储单精度浮点数	$\text{Mem}[40 + \text{Regs}[R3]] \leftarrow_{32} \text{Regs}[F0]_{0..31}$
SD F0, 40(R2)	存储双精度浮点数	$\text{Mem}[40 + \text{Regs}[R3]] \leftarrow_{64} \text{Regs}[F0]$
SH R3, 502(R2)	存储半字	$\text{Mem}[502 + \text{Regs}[R2]] \leftarrow_{16} \text{Regs}[R3]_{48..63}$
SB R2, 41(R3)	存储字节	$\text{Mem}[41 + \text{Regs}[R3]] \leftarrow_8 \text{Regs}[R2]_{56..63}$

图 B.23 MIPS的载入和存储指令。都只用一种寻址方式,并且要求存储器的值必须对齐。当然,载入和存储指令对所有的数据类型都是有效的

为了理解这些图,我们需要介绍一些C描述语言的扩展:

- 当被传送的数据长度不确切时,在符号 \leftarrow 上附加一个下标,这样 \leftarrow_n 表示传送 n 位。 $x, y \leftarrow z$ 表示 z 要传送到 x 和 y 。
- 下标用于标识字段中特定的位。位从以0开始的最高位开始标注。下标可以是一个数字(例如 $\text{Regs}[R4]_0$,表示R4的符号位),也可以是一个范围(例如 $\text{Regs}[R3]_{56..63}$ 表示R3的最低位字节)。
- 变量Mem用来表示主存储器(内存),按字节编址,可以传输任意字节的数据。
- 上标用来表示对字段进行复制(例如 0^{48} 表示一个48位长的全0字段)。
- 符号##用来链接两个字段并且可以出现在数据传送的任何一边。

例如,假设R8和R10是64位的寄存器:

$\text{Regs}[R10]_{32..63} \leftarrow_{32} (\text{Mem}[\text{Regs}[R8]])_0^{24} \text{ ## Mem}[\text{Regs}[R8]]$

意思是以R8的内容作为地址访问存储器,得到的字节按符号位扩展为32位后存入R10的低位(R10的高位不变)

所有的ALU指令都是寄存器-寄存器指令。图B.24给出了一些算术和逻辑指令的例子,包括简单的算术和逻辑操作:加、减、与、或、异或和移位。所有这些指令都支持立即寻址方式,它带

一个16位的符号扩展立即数。LUI(载入高位立即数)操作将立即数载入到寄存器的32至47位,其余位置零。这使得一个32位的常数可以用两条指令,或者通过在一条额外指令中的32位常数地址进行数据传输来得到。

指令举例	指令名称	含义
DADDU R1, R2, R3	无符号加	$\text{Regs}[R1] \leftarrow \text{Regs}[R2] + \text{Regs}[R3]$
DADDIU R1, R2, #3	加无符号立即数	$\text{Regs}[R1] \leftarrow \text{Regs}[R2] + 3$
LUI R1, #42	载入立即数到高位	$\text{Regs}[R1] \leftarrow 0^{32} \#42\#0^{16}$
DSLL R1, R2, #5	逻辑左移立即数	$\text{Regs}[R1] \leftarrow \text{Regs}[R2] \ll 5$
DSLT R1, R2, R3	置小于	if ($\text{Regs}[R2] < \text{Regs}[R3]$) $\text{Regs}[R1] \leftarrow 1$ else $\text{Regs}[R1] \leftarrow 0$

图 B.24 MIPS 中算术和逻辑指令的例子, 带有或者不带有立即数

如上面所述, R0 被用来合成最常用的操作。载入一个常数的操作可以由一个立即数和一个源操作数是 R0 的加法来实现。寄存器-寄存器传送可以通过其中一个源操作数是 R0 的加法来完成[我们有时用助记符 L1(代表载入立即数)来指前者而用 MOV 来指后者]。

MIPS 控制流指令

MIPS 也有比较指令, 比较两个寄存器的值。如果第一个寄存器的值小于第二个的值, 则比较指令将置目标寄存器为 1(代表真), 否则将置为 0(代表假)。由于这些操作都设置寄存器, 因此它们称为置等于、置不等于和置小于, 等等。同时这些比较指令也具有立即数的形式。

控制由一组跳转与一组分支来处理。图 B.25 给出了几个典型的分支和跳转指令。通过指定目标地址的两种方式及是否进行链接来区分四种跳转指令。有两种跳转指令通过把 26 位位移量移位 2 位然后覆盖程序计数器(代表紧跟转指令的一系列指令)低 28 位的方法来确定目标地址。另外两种跳转指令直接指定包含目标地址的寄存器。有两种跳转: 简单跳转和跳转并链接(用于过程调用)。后者把返回地址——下一个顺序指令的地址——放入寄存器 R31。

指令举例	指令名称	含义
J name	跳转	$\text{PC}_{36..63} \leftarrow \text{name}$
JAL name	跳转并链接	$\text{Regs}[R31] \leftarrow \text{PC} + 8; \text{PC}_{36..63} \leftarrow \text{name};$ $((\text{PC} + 4) - 2^{27}) \leq \text{name} < ((\text{PC} + 4) + 2^{27})$
JALR R2	寄存器跳转并链接	$\text{Regs}[R31] \leftarrow \text{PC} + 8; \text{PC} \leftarrow \text{Regs}[R2]$
JR R3	寄存器跳转	$\text{PC} \leftarrow \text{Regs}[R3]$
BEQZ R4, name	等于零时分支	if ($\text{Regs}[R4] == 0$) $\text{PC} \leftarrow \text{name};$ $((\text{PC} + 4) - 2^{27}) \leq \text{name} < ((\text{PC} + 4) + 2^{17})$
BNE R3, R4, name	不等于零时分支	if ($\text{Regs}[R3] \neq \text{Regs}[R4]$) $\text{PC} \leftarrow \text{name};$ $((\text{PC} + 4) - 2^{17}) \leq \text{name} < ((\text{PC} + 4) + 2^{17})$
MOVZ R1, R2, R3	等于零时转换	if ($\text{Regs}[R3] == 0$) $\text{Regs}[R1] \leftarrow \text{Regs}[R2]$

图 B.25 典型的 MIPS 控制流指令。所有的控制指令, 除了以寄存器中的地址为目标地址进行的跳转以外, 都是 PC 相对的。分支转移的距离通常超过了地址字段所能提供的距离; 由于 MIPS 指令是 32 位长, 所以把字节分支地址乘 4 以得到较长的地址

所有的分支都是条件的。分支条件由指令确定, 可能是测试源寄存器是否为零, 寄存器可能含有一个数据或者比较结果。有的分支指令判断寄存器是否为负或比较两个寄存器是否相等。分支的目标地址由 16 位带符号位移量左移 2 位后和程序计数器相加的结果来决定。还有浮点条件分支指令, 该指令通过测试浮点状态寄存器来决定是否分支。

第2章和附录A会讲到条件分支对流水线的执行有很大影响。因此许多系统结构增加了专门的指令用来将简单的分支指令转化为条件算术指令。MIPS增加了判断是否为零的条件转移。目标寄存器的值是不变还是来自于其中一个源寄存器,这将取决于另一个源寄存器的值是否为零。

MIPS的浮点数操作

浮点指令对浮点数寄存器进行操作,并指出将被使用的操作数是单精度(SP)还是双精度(DP)。MOV.S和MOV.D分别把一个单精度和一个双精度浮点数寄存器的值复制到另一个同类型的寄存器中。MFC1, MTC1, DMFC1和DMTC1在一个单精度或双精度浮点数寄存器和一个定点寄存器之间传送数据。把一个双精度数据移到两个定点寄存器中需要两条指令。还提供了定点与浮点数之间相互转换的指令。

浮点操作包括加、减、乘、除。后缀S表示单精度浮点数,而后缀D表示双精度浮点数(例如ADD.D, ADD.S, SUB.D, SUB.S, MUL.D, MUL.S, DIV.D, DIV.S)。浮点数比较指令会设置浮点状态寄存器中的某一位,可以用两条分支指令BCIT(为真分支)和BCIF(为假分支)测试状态寄存器来决定是否进行分支。

为了提高图形处理性能,MIPS64提供了在一个64位浮点数寄存器两半部分中分别进行两个32位浮点数的操作。这种配对单独操作包括:ADD.PS, SUB.PS, MUL.PS和DIV.PS,都采用双精度浮点数的载入存储指令进行载入和存储。

MIPS64也认识到了多媒体应用程序的重要性,设置了定点和浮点数的乘加指令:MADD, MADD.S, MADD.D和MADD.PS。这些组合操作中的寄存器大小相等。图B.26列出了MIPS64操作的一个子集及其意义。

指令类型/操作码	指令含义
数据传输	在寄存器与存储器之间传送数据,或者是在定点寄存器与浮点寄存器或专用寄存器之间传送数据;存储器唯一的寻址方式是16位移量+GPR的内容
LB, LBU, SB	载入字节, 载入无符号字节, 存储字节(均为定点寄存器操作)
LH, LHU, SH	载入半字, 载入无符号半字, 存储半字(均为定点寄存器操作)
LW, LWU, SW	载入字, 载入无符号字, 存储字(均为定点寄存器操作)
LD, SD	载入SP浮点数, 载入DP浮点数, 存储SP浮点数, 存储DP浮点数
L.S, L.D, S.S, S.D	在GPR与专用寄存器之间传送数据
MFC0, MTC0	复制一个SP或DP寄存器的内容到另一个FP寄存器
MOV.S, MOV.D	在定点寄存器和FP寄存器之间传送32位数据
MFC1, MTC1	在GPR中的定点或逻辑数据的操作;有符号数在溢出时产生陷阱中断
算术/逻辑	加, 加立即数(所有立即数都是16位), 加符号数, 加无符号数
DADD, DADDI, DADDU, DADDIU	减, 有符号减, 无符号减
DSUB, DSUBU	乘和除, 有符号和无符号乘除;乘加;所有的操作都使用并产生64位的值
DMUL, DMULU, DDIV, DDIVU, MADD	与, 立即数与
AND, ANDI	或, 立即数或, 异或, 立即数异或
OR, ORI, XOR, XORI	载入立即数到高位, 载入立即数到寄存器的32位到47位, 并将其符号展开
LUI	移位, 立即数形式(DS_), 变量形式(DS_V);逻辑左移, 逻辑右移, 算数右移
DSLL, DSRL, DSRA, DSRLV, DSRLV, DSRAV	置小于, 小于立即数;有符号和无符号
SLT, SLTI, SLTU, SLTIU	

图B.26 部分MIPS64指令列表。这些指令的格式如图B.22所示。SP表示单精度, DP表示双精度, 在本书最后的附页中也可以找到本表

指令类型/操作码	指令含义
控制	条件转移或跳转; PC 相对或者通过寄存器指明跳转地址
BEQZ, BNEZ	GPR 为零或不为零时跳转; 相对于 PC + 4 的 16 位偏移
BEQ, BNE	GPR 相等或不等时跳转; 相对于 PC + 4 的 16 位偏移
BC1T, BC1F	测试 FP 状态寄存器中的比较位并转移; 相对于 PC + 4 的 16 位偏移
MOVN, MOVZ	如果一个 GPR 为负或零, 复制另一个 GPR 到第三个 GPR
J, JR	跳转; 相对于 PC + 4 的 26 位偏移 (J) 或者由寄存器指定 (JR)
JAL, JALR	跳转交链接; 把 PC + 4 保存在 R31, 目标地址为相对于 PC (JAL) 或是寄存器指定 (JALR)
TRAP	以一个向量地址传输给操作系统
ERET	从异常中返回用户代码, 恢复用户模式
浮点操作	DP 和 SP 格式的 FP 操作
ADD.D, ADD.S, ADD.PS	加 DP, SP 浮点数; 配对 SP 加
SUB.D, SUB.S, SUB.PS	减 DP, SP 浮点数; 配对 SP 减
MUL.D, MUL.S, MUL.PS	乘 DP, SP 浮点数; 配对 SP 乘
MADD.D, MADD.S, MADD.PS	DP, SP, 配对 SP 乘加
DIV.D, DIV.S, DIV.PS	除 DP, SP 浮点数; 配对 SP 除
CVT.L, CVT.S	转换指令: CVT.x.y 把 x 转换成 y, 其中 x 和 y 是 L (64 位定点), W (32 位定点), D (双精度浮点数) 或 S (单精度浮点数), 所有操作数都是 FPR
C.L.D, C.L.S	DP 和 SP 比较: “_” = LT, GT, LE, GE, EQ, NE; 设置 FP 状态寄存器中的位

图 B.26 (续) 部分 MIPS64 指令列表。这些指令的格式如图 B.22 所示。SP 表示单精度, DP 表示双精度, 在本书最后的附页中也可以找到本表

MIPS 指令系统的使用

为了说明哪些指令更常用, 图 B.27 给出了 5 个 SPECint2000 程序的指令类别和指令调用频率。图 B.28 给出了 5 个 SPECfp2000 程序的指令类别和指令调用频率。

指令	gap	Gcc	gzip	Mcf	Perl	定点平均值
load	26.5%	25.1%	20.1%	30.3%	28.7%	26%
store	10.3%	13.2%	5.1%	4.3%	16.2%	10%
add	21.1%	19.0%	26.9%	10.1%	16.7%	19%
sub	1.7%	8.2%	5.1%	3.7%	8.5%	3%
mul	1.7%	0.1%				0%
compare	8.8%	6.1%	6.6%	6.3%	3.8%	5%
load imm	4.8%	8.5%	1.5%	0.1%	1.7%	2%
cond branch	9.3%	18.1%	11.0%	1.5%	10.9%	12%
cond mov	0.4%	0.6%	1.1%	0.1%	1.9%	1%
jump	0.8%	0.7%	0.8%	0.7%	1.7%	1%
call	1.6%	0.6%	0.4%	3.2%	1.1%	1%
return	1.6%	0.6%	0.4%	3.2%	1.1%	1%
shift	3.8%	1.1%	8.1%	1.1%	0.5%	2%
and	4.3%	4.6%	9.4%	0.2%	1.2%	4%
or	7.9%	8.5%	4.8%	7.6%	8.7%	9%
xor	1.8%	8.1%	4.4%	1.5%	8.8%	3%
other logical	0.1%	0.4%	0.1%	0.1%	0.3%	0%
load FP						0%
store FP						0%

图 B.27 5 个 SPECint2000 程序中各类 MIPS 指令所占的比例。定点寄存器-寄存器传送指令包括在 or 指令中。空的表项值为 0.0%

指令	gap	Gcc	gzip	Mcf	Perl	定点平均值
						0%
add FP						0%
sub FP						0%
mul FP						0%
div FP						0%
mov reg-reg FP						0%
compae FP						0%
cond mov FP						0%
other FP						0%

图 B.27 (续) 5 个 SPECint2000 程序中各类 MIPS 指令所占的比例。定点寄存器 - 寄存器传送指令包括在 or 指令中。空的表项值为 0.0%

指令	applu	art	equake	lucas	swim	浮点平均值
load	13.8%	18.1%	22.3%	10.6%	9.1%	15%
store	2.9%		0.8%	3.4%	1.3%	2%
add	30.4%	30.1%	17.4%	11.1%	24.4%	23%
sub	2.5%		0.1%	2.1%	3.8%	2%
mul	2.3%			1.2%		1%
compare		7.4%	2.1%			2%
load imm	13.7%		1.0%	1.8%	9.4%	5%
cond branch	2.5%	11.5%	2.9%	0.6%	1.3%	4%
cond mov		0.3%	0.1%			0%
jump			0.1%			0%
call			0.7%			0%
return			0.7%			0%
shift	0.7%		0.2%	1.9%		1%
and			0.2%	1.8%		0%
or	0.8%	1.1%	2.3%	1.0%	7.2%	2%
xor		3.2%	0.1%			1%
other logical			0.1%			0%
load FP	11.4%	12.0%	19.7%	16.2%	16.8%	15%
store FP	4.2%	4.5%	2.7%	18.2%	5.0%	7%
add FP	2.3%	4.5%	9.8%	8.2%	9.0%	7%
sub FP	2.9%		1.3%	7.6%	4.7%	3%
mul FP	8.6%	4.1%	12.9%	9.4%	6.9%	8%
div FP	0.3%	0.6%	0.5%		0.3%	0%
mov reg-reg FP	0.7%	0.9%	1.2%	1.8%	0.9%	1%
compae FP		0.9%	0.6%	0.8%		0%
cond mov FP		0.6%		0.8%		0%
other FP				1.6%		0%

图 B.28 5 个 SPECint2000 程序中各类 MIPS 指令所占的比例。定点寄存器 - 寄存器传送指令包括在 or 指令中。空的表项值为 0.0%

B.10 谬误和易犯的错误

系统结构设计者经常会陷入一些很常见的错误想法中。在这一节中，我们将讨论一些例子。

易犯的错误：设计一种“高级”指令系统特性就意味着要支持一种高级语言结构。

试图将高级语言的特征并入指令系统中使得设计者们设计出一些灵活性很强、强有力的指令。但是,这些指令经常会做些需求以外的工作,或者不能准确地符合一些语言的要求。曾经有很多的努力花在20世纪70年代称为“语义缺口”的问题上。虽然出发点是想补充指令系统使得硬件的功能达到语言的水平,但是,这些增加的功能却又导致了被Wulf[1981]称为语义冲突的问题:

……由于给指令赋予了过多内容,因此机器的设计者仅能够在有限的环境中使用指令。[P.43]

对于经常发生的情况来说,这种指令的功能通常过于强大,这就导致了许多不必要的工作以及指令速度的降低。VAX的CALLS指令就是一个很好的例子。CALLS指令采用了一种“被调用者保存”策略(要保存的寄存器由被调用者决定),但是保存的动作由调用者的调用指令完成。CALLS指令首先把参数压入堆栈,然后完成以下几个步骤:

1. 如果有必要,将堆栈对齐。
2. 把参数个数压入堆栈。
3. 把过程调用屏蔽码所指出的寄存器压入堆栈(见B.11节)。这个屏蔽码保存在被调用的过程代码中——这使得被调用者可以指定调用者要保存的寄存器,即使在分开编译的情况下也是如此。
4. 把返回地址压入堆栈,然后再压栈顶与栈底的指针(作为启动记录)。
5. 清除条件码,它会吧允许陷阱设为已知状态。
6. 压入一个状态字,再压入一个零字。
7. 更新两个堆栈指针。
8. 分支到过程的第一条指令。

实际程序中绝大多数的调用并不需要这么多的开销。大多数过程知道它们的参数个数,可以通过使用寄存器而不是堆栈来传递参数以建立一种更快的链接协定。此外,CALLS指令强制使用两个寄存器用做链接,而很多语言只需要一个链接寄存器。很多支持过程调用和活动栈管理的尝试都不能成功地应用,要么是因为它们没有达到语言的要求,要么是因为它们太通用化而使得应用过于昂贵。

VAX的设计者们提供了一个简单的指令:JSB,它的速度相对较快,因为它只将返回的PC值压入栈然后就跳转至过程。然而大多数VAX的编译器使用代价更大的CALLS指令。CALLS指令被包含在系统结构中,是为了让过程的链接协议标准化。其他的机器通过与编译器设计者之间达成一致的方法来对调用协议进行标准化,而不需要一个复杂且非常通用的过程调用指令。

谬误:存在一种典型的程序。

很多人都倾向于相信存在一个典型的程序,可以用它来设计一个理想的指令系统。我们可以参考在第1章中所讨论的综合基准测试程序,其中的数据清楚地表明程序在使用指令系统方面存在显著的差别。例如,图B.29所示为在四个SPEC2000的程序中数据传输大小的情况:很难说这四个程序中哪一个是典型的。对于专门支持一类应用的指令系统,这种差异可能会更大,例如十进制指令在其他应用中就不会被使用。

易犯的错误:可以不考虑编译器而改进指令系统以缩减代码大小。

图B.30显示的是MIPS指令系统下四个编译器产生的相对代码大小。尽管设计师努力使代码减少了30%~40%,不同的编译器策略却更大程度地影响着代码的大小。就像性能优化技术一样,在试图改进硬件以节约空间之前首先应考虑编译器如何产生较少的代码。

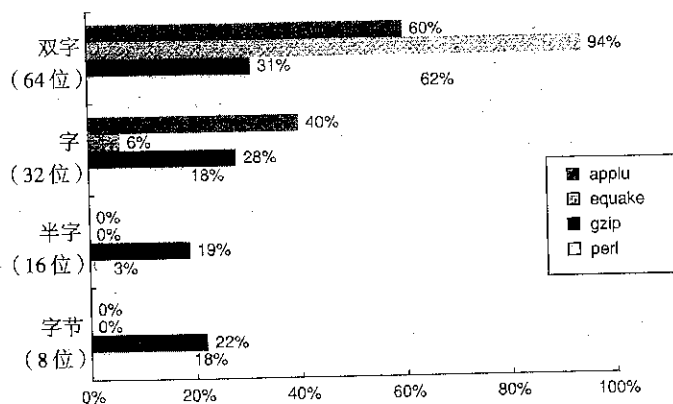


图 B.29 四个SPEC2000程序引用数据的大小。虽然可以计算一个平均值,但是无法下结论说该平均值就是一个典型程序

编译器	Apogee software Version 4.1	Green hills Multi2000 v 2.0	Algorithmics Sde4.0b	ldt/c 7.2.1
系统结构	MIPS IV	MIPS IV	MIPS 32	MIPS 32
处理器	NEC VR5432	NEC VR5000	IDT 32334	IDT 79RC32364
自相关内核	1.0	2.1	1.1	2.7
卷积编码器内核	1.0	1.9	1.2	2.4
定点位分配内核	1.0	2.0	1.2	2.3
定点复数FFT内核	1.0	1.1	2.7	1.8
Viterbi GSM 译码器内核	1.0	1.7	0.8	1.1
5个内核的几何平均值	1.0	1.7	1.4	2.0

图 B.30 EENBC 的通信程序在不同编译器下相对于 Apogee Software V4.1 C 编译器的代码大小。指令集系统结构基本一样,但代码大小相差最多达一倍。这些数据来自于 2000.2~2000.6 的报告

谬误: 有缺陷的系统结构不可能是一种成功的系统结构。

80x86 给我们提供了一个很明显的例子: 它的结构只有它的设计者才喜欢(见附录 J)。Intel 的工程师们试图更正 80x86 设计中不受欢迎的设计。例如, 80x86 支持段式存储, 而所有其他的机器都选择了页式存储; 80x86 对整型数据使用扩展的累加器, 而其他的机器使用通用寄存器; 80x86 使用一个堆栈来保存浮点数据, 而所有其他的机器在很久以前就废弃了执行堆栈。

尽管存在这些缺陷, 80x86 系统结构还是获得了巨大的成功。这主要有三方面的原因: 首先, 它被选为 IBM PC 的微处理器, 使得在二进制上与 80x86 兼容变得非常重要; 其次, 摩尔定律保证了有足够的资源可使 80x86 在内部转化为 RISC 指令系统, 然后执行类 RISC 指令。这两点使得与它大量 PC 软件二进制兼容, 在性能上和 RISC 处理器不相上下; 最后, 大量的处理器出货使 Intel 有能力支付这种高代价的硬件转化。此外, 大量的处理器出货使生产厂家可以跟得上学习曲线, 这也在一定程度上降低了成本。

对嵌入式应用程序来说, 为了转换而增大内核尺寸并增加功耗很不合适, 但对桌面应用而言这样做市场意义很大, 其性价比也受到了服务器应用的关注, 唯一的弱点在于其地址是 32 位的, 但这一弱点在 AMD64 的 64 位地址中已经得以解决(见第 5 章)。

谬误: 可以设计一个没有缺陷的系统结构。

所有系统结构的设计都涉及一系列硬件和软件技术之间的折中。这些技术可能会随着时间的发展而发生变化,那些在设计者当初看来好像是正确的决定事后可能证明是错误的。例如,在1975年,VAX的设计者们过分强调了代码大小的重要性,而没有估计到5年后译码的简化和流水线是那么重要。RISC系统中一个典型的例子就是延迟分支(见附录J)。对于5级流水线的处理器来说这很容易解决,但如果处理器的流水线更长,在一个时钟周期内同时执行数条指令其困难就比较大。此外,几乎所有的系统结构最终都会产生地址空间不足的问题。

然而,为了避免这些长期的问题,意味着需要系统结构目前的效率做出妥协,这是很冒险的,因为新的指令系统必须在开始的一段时间历经考验而生存下来。

B.11 结论

最早的系统结构中指令系统在很大程度上受硬件技术制约。一旦硬件技术允许,设计者们就开始寻找支持高级语言的方法。这导致了在计算机发展过程中的三个阶段。在20世纪60年代,堆栈系统结构日益流行起来。它们被认为能够最好地配合高级语言——就当时的编译器技术而言,这或许是对的。到了20世纪70年代,设计的主要焦点是如何降低软件的费用,具体主要是用硬件来取代软件,或是提供能够简化软件设计者工作的高级结构。结果是:既产生了高级语言计算机系统结构,又出现了诸如VAX一类的强大系统结构。VAX具有大量的寻址方式、多种数据类型以及一个高度正交化的系统结构。20世纪80年代,更加成熟的编译技术以及重新强调机器效率的观点使得更加简单的系统又成为了主流,它主要基于load-store系统结构的机器。

20世纪90年代指令集系统结构发生了一些变化:

- **地址空间翻倍:** 32位的地址指令系统扩展到64位,寄存器的位数扩充到64位。附录C给出了三种从32位扩展到64位的系统结构的例子。
- **通过条件执行实现条件分支转移的优化:** 在第2章和第3章我们会看到条件分支严重地限制了成功设计对性能的提升。因此用条件操作完成来代替条件分支有很大的好处,例如条件传送(见附录G)已经被大多数指令系统采用。
- **通过预取优化Cache性能:** 第5章指出了存储器层次结构在提高机器性能方面越来越重要的地位,Cache的一次不命中所花掉的时间与早期机器中一次缺页错误所花掉的时间一样多,因此增加了预取指令来减少Cache不命中的代价。
- **对多媒体的支持:** 大多数桌面和嵌入式指令系统都增加了对多媒体和DSP应用程序的支持,这在本附录有所介绍。
- **更快的浮点操作:** 附录I给出了一些提高浮点性能的新操作。如执行一个乘法和一个加法的操作,执行配对单精度的操作。我们结合MIPS讨论这些内容。

在1970年到1985年之间,很多人都认为计算机系统结构设计者的主要任务就是设计指令系统。其结果是,当时的系统结构教科书都在强调指令系统设计,这和20世纪50年代和60年代中的系统结构教科书强调算术运算的情况类似。通常认为受过专业训练的系统结构设计者们应该对流行机器的优缺点有一定的了解。在对指令系统设计的改进过程中,二进制兼容性没有得到研究人员和教科书作者的重视,这就给人留下了这样一个印象:许多设计者都有机会设计一个指令系统。

今天计算机系统结构的定义已经扩展为包括对全部计算机系统的设计和评价,而不仅仅是定义指令系统。因而对于设计者们来说,还有很多的课题需要研究。事实上,本附录的内容是本书1990年版的一个核心方面,但是现在主要作为相关材料包括在附录中。

附录J可能会使关注指令集系统结构的读者更感兴趣:它描述了各种指令系统,它们或者如今仍然适用,或者在技术发展过程中具有重要地位,附录J还通过MIPS比较了9种流行的load-store计算机。

B.12 历史回顾和参考文献

随书光盘上的K.3节的特色是关于指令系统演变的讨论,且包括进一步阅读和开发相关主题的参考文献。

是定义
书 1990

或者如
ad-store

主题的

附录 C 存储器层次结构回顾

Cache: 用来隐藏和存储信息的安全位置。

Webster's New World Dictionary of the
American Language
Second College Edition(1976)

C.1 简介

本附录将对存储器层次结构做个简要回顾, 包括 Cache 的基本内容、虚拟存储器、性能公式以及简单优化。本节内容涵盖了以下 36 个术语:

高速缓存 (Cache)	全相联 (fully associative)	写分配 (write allocate)
虚拟存储器 (virtual memory)	重写 (脏) 位 (dirty bit)	一体 Cache (unified Cache)
存储器停顿周期 (memory stall cycles)	块内偏移 (block offset)	每条指令缺失次数 (misses per instruction)
直接映射 (direct mapped)	写回法 (write back)	块 (block)
有效位 (valid bit)	数据 Cache (data Cache)	局部性 (locality)
块地址 (block address)	命中时间 (hit time)	地址跟踪 (address trace)
写直达 (write through)	Cache 缺失 (Cache miss)	组 (set)
指令 Cache (instruction Cache)	页缺失 (page fault)	随机替换 (random replacement)
平均存储器访问时间 (average memory access time)	缺失率 (miss rate)	索引字段 (index field)
Cache 命中 (Cache hit)	n 路组相联 (n-way set associative)	不按写分配 (no-write allocate)
页 (page)	最近最少使用 (least-recently used)	写缓存 (write buffer)
缺失代价 (miss penalty)	标志字段 (tag field)	写停顿 (write stall)

有关这部分的更详细内容, 可参考 *Computer Organization and Design* 一书的第 7 章, 它是专门为初学者而编写的。

Cache 通常是存储器层次结构中第一层的名字, 是距离处理器最近的存储层次。由于局部性原理已广泛应用到计算机系统设计的诸多方面, 利用其优点来提高系统性能是普遍采用的方法, 因此, 术语 Cache 如今被应用到了任何通过缓冲来重复利用经常发生事件的情形中; 这些例子包括文件 Cache、命名 Cache 等。

当处理器在 Cache 中找到要访问的数据项时, 称之为 Cache 命中; 当处理器找不到所需要的数据项时, 称之为 Cache 缺失。包含所需字的固定大小的数据集称为一个块, 它来源于存储器并被放置于 Cache 中。时间局部性表明, 该字很可能在不久的将来再次被用到, 所以将它放到处理器能很快访问到的 Cache 中是非常有意义的。根据空间局部性原理, 块内的其他数据也可能在不久的将来被访问到。

发生 Cache 缺失时, 数据访问时间取决于存储器的时延和带宽。时延决定了读取块第一个字的时间, 而带宽决定了读取该块的其他部分所需要的时间。Cache 缺失通常由硬件处理, 它会导致顺

序执行处理器暂停或停止处理,直到所需数据可用为止。当乱序执行时,欲使用结果的指令在Cache缺失时仍然要等待,但是其他的指令可以继续执行。

与此类似,一个程序需要的所有数据并不是都驻留在内存中。如果系统中包含虚拟存储器,那么一些数据可以存放在磁盘中。地址空间通常被划分为一些大小固定的块,称之为“页”。在任意时刻,每一页保存在内存或者磁盘中。当处理器要访问的数据项既不在Cache中又不在内存中时,则发生“页缺失”,需要从磁盘中取出该页并放到内存中。页缺失花费的时间比较长,一般由软件来处理,这时,处理器并不停止工作,而是切换到其他任务继续执行。Cache与内存的关系和内存与磁盘的关系相同。

图C.1给出了从高端桌面处理器到低端服务器的各类计算机系统中,存储器层次结构中第一级的容量和访问时间范围。

层次	1	2	3	4
名称	寄存器	Cache	内存	磁盘
典型容量	< 1 KB	< 16 MB	< 512 GB	> 1 TB
实现技术	多端口定制存储器, CMOS	片内或片外 CMOS SRAM	CMOS DRAM	磁盘
访问时间 (ns)	0.25~0.5	0.5~25	20~250	5 000 000
带宽 (MB/s)	50 000~500 000	5000~20 000	2500~10 000	50~500
管理	编译器	硬件	操作系统	操作系统/操作者
下一级	Cache	内存	磁盘	CD 或磁带

图 C.1 大型工作站和小型服务器中的典型存储层次结构: 距离处理器越远,存储速度越慢、容量越大。嵌入式计算机可以没有磁盘存储,并且具有小得多的存储器和Cache。访问时间随着层次结构从高层到低层依次增加,这种结构对响应频率数据传输的管理是合理的。“实现技术”一行给出了实现这些功能的典型技术。访问时间给出的是2006年典型的纳秒级数据,这些时间还会进一步减少。存储层次各级间的带宽以MB/s为单位。磁盘存储器的带宽包括存储介质和缓冲区接口

Cache 性能回顾

根据局部性原理和小存储器具有较快访问速度的原理可知,存储器层次结构能显著改进计算机系统的性能。我们可以扩展第1章中的处理器执行时间公式来评估Cache的性能改进程度。通常将处理器暂停工作、等待一次存储器访问的周期数称为存储器停顿周期数,性能可以表示为处理器时钟周期数与存储器停顿周期数的和,再乘以时钟周期时间:

$$\text{CPU 执行时间} = (\text{CPU 时钟周期数} + \text{存储器停顿周期数}) \times \text{时钟周期时间}$$

等式中假定CPU时钟周期数包含了处理Cache命中和Cache缺失时处理器停顿的时间。C.2节将会重新探讨这个简化的假设。

存储器停顿周期数由缺失次数和每一次的缺失代价所决定:

$$\begin{aligned}
 \text{存储器停顿周期数} &= \text{缺失次数} \times \text{缺失代价} \\
 &= \text{执行指令数} \times \frac{\text{缺失次数}}{\text{指令数}} \times \text{缺失代价} \\
 &= \text{执行指令数} \times \frac{\text{存储器访问次数}}{\text{指令数}} \times \text{缺失率} \times \text{缺失代价}
 \end{aligned}$$

最后一种表示形式的优点是每一个因子都很容易得到。我们已经知道如何计算执行指令数(对于测试型处理器,我们只计算最终提交的指令),测定每条指令的存储器访问次数可用同样的方式;每条指令都需要一次取指令,并且很容易判断出该指令是否会访问数据。

注意,此前我们把缺失代价作为平均数进行计算,但在后面的讨论中我们将它看做是一个常数。当发生Cache缺失时,Cache下一层次的存储器由于上一层次存储器的请求和存储器刷新而变得异常忙碌。并且在处理器、总线和存储器之间相互传输数据时,时钟周期数也是不同的。因此,通常把缺失代价作为一个常量处理是一种简化。

缺失率是Cache访问中产生缺失所占访问总数的百分比(也就是产生Cache缺失的访问次数除以访问总次数)。缺失率可以用Cache仿真器来测量:仿真器对要进行访问的指令和数据进行地址跟踪,用Cache仿真行为来判断是命中还是缺失,并给出命中和缺失的总数。一些微处理器提供了硬件来计算缺失次数和存储器访问次数,这使得缺失率的计算变得更为简单和快速。

读、写Cache操作的缺失率和缺失代价往往是不同的,因此上面的公式只是一个近似公式。存储器停顿周期数可以由每条指令访问存储器的次数、读和写的缺失代价(以时钟周期数计)和缺失率来定义:

$$\text{存储器停顿周期数} = \text{执行指令数} \times \text{每条指令读次数} \times \text{读缺失率} \times \text{读缺失代价} + \text{执行指令数} \times \text{每条指令写次数} \times \text{写缺失率} \times \text{写缺失代价}$$

可以将读、写合并,计算出它们的平均缺失率和缺失代价,从而将上述公式简化为

$$\text{存储器停顿周期数} = \text{执行指令数} \times \frac{\text{存储器访问次数}}{\text{指令数}} \times \text{缺失率} \times \text{缺失代价}$$

缺失率是Cache设计中最重要指标之一,然而,在后续的章节中将会看到它并不是唯一的指标。

例题 假定有一台计算机,当所有存储器访问操作都能在Cache中命中时,每条指令的时钟周期数(CPI)为1.0。数据访问只包含load和store,这些指令占全部指令的50%。缺失代价为25个时钟周期,缺失率为2%,当所有指令都在Cache中命中时,计算机性能提高多少?

解答: 首先计算Cache始终命中时的机器性能:

$$\begin{aligned} \text{CPU 执行时间} &= (\text{CPU 时钟周期数} + \text{存储器停顿周期数}) \times \text{时钟周期} \\ &= (\text{执行指令数} \times \text{指令执行时钟数} + 0) \times \text{时钟周期} \\ &= \text{执行指令数} \times 1.0 \times \text{时钟周期} \end{aligned}$$

再看实际Cache的操作情况,首先计算存储器停顿周期数:

$$\begin{aligned} \text{存储器停顿周期数} &= \text{执行指令数} \times \frac{\text{存储器访问次数}}{\text{指令数}} \times \text{缺失率} \times \text{缺失代价} \\ &= \text{执行指令数} \times (1 + 0.5) \times 0.02 \times 25 \\ &= \text{执行指令数} \times 0.75 \end{aligned}$$

上式中,(1+0.5)表示每条指令包括一条存储器指令访问和0.5条存储器数据访问。因此,总性能表示如下:

$$\begin{aligned} \text{CPU 行时间}_{\text{Cache}} &= (\text{执行指令数} \times 1.0 + \text{执行指令数} \times 0.75) \times \text{时钟周期} \\ &= 1.75 \times \text{执行指令数} \times \text{时钟周期} \end{aligned}$$

性能比就是两个执行时间比值的倒数:

$$\begin{aligned}\text{CPU执行时间}_{\text{cache}} &= \frac{1.75 \times \text{指令数} \times \text{时钟周期}}{1.0 \times \text{指令数} \times \text{时钟周期}} = 1.75 \\ &= \text{执行指令数} \times 0.75\end{aligned}$$

由此可知, 不发生 Cache 缺失时, 计算机性能是原来的 1.75 倍。

一些设计者喜欢用平均每条指令缺失次数而不是平均每次存储器访问缺失数来衡量缺失率, 两者之间的关系如下:

$$\frac{\text{缺失率}}{\text{指令数}} = \frac{\text{缺失率} \times \text{存储器访问次数}}{\text{指令数}} = \text{缺失率} \times \frac{\text{存储器访问次数}}{\text{指令数}}$$

当知道每条指令平均访问存储器次数时, 可以将缺失率转换为每条指令缺失次数, 此时后一个公式更为有用。例如, 我们可以将上例中的访存缺失率转换成每条指令缺失次数:

$$\frac{\text{缺失率}}{\text{指令数}} = \text{缺失率} \times \frac{\text{存储器访问次数}}{\text{指令数}} = 0.02 \times 1.5 = 0.030$$

通常情况下, 为了将每条指令的缺失次数表示为整数, 我们一般采用每千条指令的缺失次数来计算。这样, 上面的答案也可表示成每千条指令发生缺失 30 次。

这种方法的优点是它独立于硬件实现。例如, 推测处理器的指令预取单元取到的指令条数为实际提交的 2 倍, 如果采用每次存储器访问缺失数而不是每条指令缺失数来衡量缺失率, 就会人为地降低缺失率。这种方法的缺点是平均每条指令缺失次数与处理器系统结构相关, 例如, 平均每条指令存储器访问次数, 对于 80x86 和 MIPS 来说可能是完全不同的。总之, 尽管 RISC 系统结构的相似性允许我们参照其他系统结构的参数, 但系统结构设计师一般在单一计算机系列的情况下会采用每条指令缺失次数。

例题 为表明两个缺失率等式的等价关系, 我们重新看一下上面的例子。这次假定缺失率为每千条指令缺失 30 次, 以指令数计算的存储器停顿周期是多少?

解答: 重新计算存储器停顿周期:

$$\begin{aligned}\text{存储器停顿周期} &= \text{缺失次数} \times \text{缺失代价} \\ &= \text{执行指令数} \times \frac{\text{存储器访问次数}}{\text{指令数}} \times \text{缺失代价} \\ &= \text{执行指令数} / 1000 \times \frac{\text{存储器访问次数}}{\text{指令数} / 1000} \times \text{缺失代价} \\ &= \text{执行指令数} / 1000 \times 30 \times 25 \\ &= \text{执行指令数} / 1000 \times 750 \\ &= \text{执行指令数} \times 0.75\end{aligned}$$

我们得到的答案与上面的相同。

存储器层次结构的四个问题

通过回答四个基本问题, 我们继续对存储器层次结构的第一层次——Cache 进行介绍。

Q1: 块的放置。在较高层中, 一个块可以被放置在哪里?

Q2: 块的标志。如果一个块在较高层中, 如何找到它?

Q3: 块的替换。如果块发生缺失, 哪个块应该被替换?

Q4: 写时策略。写操作时会发生什么?

对这些问题的答案能帮助我们理解不同层次存储器之间的折中；因此，我们对每个例子都提出这四个问题。

Q1：一个块可以被放置到 Cache 的什么地方？

图 C.2 给出了根据块放置策略来划分的三种 Cache 组织形式：

- 如果每个块在 Cache 中只能出现在唯一位置上，那么这种映射就称为直接映射，映射方法通常是

$$(\text{块地址}) \bmod (\text{Cache 中的块数})$$

- 如果一个块可以放到 Cache 中的任何一个地方，那么这种映射就称为全相联映射。
- 如果一个块必须被严格地放置到 Cache 中某组位置里，那么这种映射就称为组相联映射。一个组是 Cache 中的一组块。一个块首先被映射到一个组中，然后它可以被放置到组中的任何一个块中。组通常利用位选择方式确定。即

$$(\text{块地址}) \bmod (\text{Cache 的组数})$$

如果一个组里有 n 块，那么这种映射方法就称为 n 路组相联。

直接映射和全相联实际上是组相联的两种特殊情况。直接映射仅仅是一个简单的 1 路组相联，而一个有 m 块的全相联 Cache 可以称为 m 路组相联，也就是说，这里直接映射可以视为有 m 组的组相联映射，而全相联是只有 1 组的组相联映射。

今天大多数处理器的 Cache 采用直接映射、2 路组相联映射或是 4 路组相联映射方式。

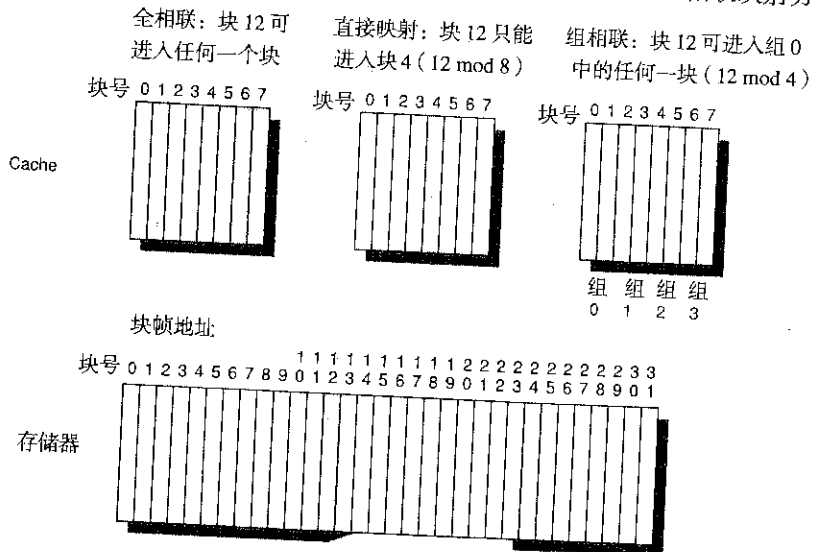


图 C.2 例中 Cache 分成 8 个块结构，存储器有 32 个块。Cache 的三种策略从左到右如上所示。假定 Cache 开始时是空的，而要分配的块地址在内存中为 12。在全相联方式中，内存块 12 可以进入到 Cache 的 8 个块中的任何一个。在直接映射方式中，内存块 12 只能放到块 4 处 (12 模 8)。组相联方式具有上述两种方式的某些特性，允许块被放置到组 0 (12 模 4) 中的任何一个位置上。因为每组两块，所以这意味着 12 可以被放置到 Cache 的块 0 中或块 1 中。真正的 Cache 包括几千个块结构，而真正的存储器则包含成百万的块。该组相联方式有 4 个组，每组两个块，因而称为 2 路组相联。在这里假定 Cache 是空的，且问题中提到的块是指低层次的块 12

Q2: 如果某一块在 Cache 中, 如何找到它?

Cache 的每一个块结构中都有一个地址标志给出块地址。查找目标信息是通过每一个 Cache 块标志进行检查, 看它是否与来自处理器的块地址相匹配来实现的。考虑到速度的重要性, 所有可能的标志都要被并行检查。

必须要有一种方法来确定一个块中是否包含有效信息。最通用的方法是在标志中加上一个有效位, 来表明该项是否包含有效地址。如果该位未被设置, 则该地址不能进行匹配。

在继续下一个问题之前, 我们先来看一下处理器地址与 Cache 地址间的关系。图 C.3 给出了 Cache 地址的划分情况: 第一级划分为块地址和块内偏移; 块地址可以进一步细分为标志字段和索引字段。块内地址偏移用来从块中选出需要的数据, 索引字段用来选择组, 通过比较标志字段来判断是否发生命中。判断是否命中时, 没有必要对除标志字段以外的更多地址位进行比较, 原因如下:

- 没有必要对块内偏移进行比较, 这是因为 Cache 命中与否的单位是整个块, 因此, 所有块内的地址偏移将会匹配。
- 由于索引字段用来选择被检查的组, 因此, 对索引字段的比较是多余的。例如, 一个地址存储在组 0 中, 则在索引字段中必然包含 0, 否则, 不可能被存储在组 0 中; 组 1 一定有索引值 1; 依此类推。这种优化通过减少 Cache 标志字段长度来节省硬件开销和功耗。

如果 Cache 的大小一定, 增加相联度将增加每一个组中的块数, 因而会减少索引字段的宽度, 但会增加标志宽度。也就是说, 图 C.3 中的标志和索引字段的界限将根据增加的相联度向右移动, 最后, 如果变成全相联, 将不存在索引字段。

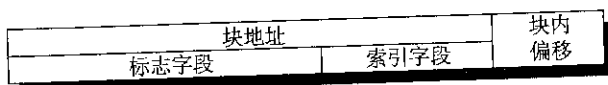


图 C.3 组相联或直接映射中地址的三个部分。标志字段用来检查组中的所有块, 而索引字段用来选择组。块内地址偏移是块内期望数据的地址。全相联映射没有索引字段

Q3: 如果 Cache 缺失, 哪个块应该被替换?

当缺失发生时, Cache 控制器必须选择 Cache 中的一个块, 并用欲获得的数据来替换它。使用直接映射替换方法的优点是硬件决策简单——事实上, 它根本不需要任何选择: 只检查一个块是否命中, 当不命中时, 也只有这个块可以用于替换。如果是全相联或是组相联, 则发生缺失时就要从多个块中做出适当的选择。有三种基本的块替换策略:

- **随机替换策略:** 为了均匀地分配, 候选块将被随机选择。一些系统产生伪随机数块号, 以获得可重复的行为, 当调试硬件时, 这种方式极其有用。
- **最近最少使用 (LRU) 替换策略:** 为了减少替换那些可能不久就要用到信息的概率, 需要记录块的访问次数。利用历史信息来预测未来使用情况, 被替换的块将是最长时间内没有被访问的 Cache 块。LRU 使用了一个局部性原理的推论: 如果一个最近被使用过的块很可能会被再次访问, 那么最好替换最近最少使用的块。
- **先进先出 (FIFO) 替换策略:** 由于计算 LRU 比较复杂, 该替换策略将最早进入 Cache 的块作为替换块。

随机替换的优点是硬件简单。当跟踪的块数增加时, LRU 硬件也随之变得更为复杂, 而且它通常仅仅是近似的。图 C.4 给出了 LRU、随机和先进先出三种替换策略在缺失率上的差别。

容量	相联度								
	2 路			4 路			8 路		
	LRU	随机替换	FIFO	LRU	随机替换	FIFO	LRU	随机替换	FIFO
16 KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	118.8	110.4
64 KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

图 C.4 LRU、随机替换和先进先出三种替换策略，在不同 Cache 大小和相联度的情况下，每千条指令缺失率的比较。这些数据是在 Alpha 系统结构计算机中，通过运行 10 个 SPEC2000 基准测试程序得到的，Cache 块大小为 64 字节。在 10 个基准测试程序中，5 个来自于 SPECint2000 (gap, gcc, gzip, mcf, perl)，另外 5 个来自于 SPECfp2000 (applu, art, equake, lucas, swim)。这里，对于大容量 Cache，LRU 和随机替换几乎没有差别；对于小容量 Cache，LRU 替换策略最优，先进先出 (FIFO) 算法要比随机算法性能略好一些。在本章后面的大多数图表中，我们都会以这个计算机系统和这些基准测试程序为例来说明

Q4：写操作时会发生什么？

读操作在处理器 Cache 的访问中占大多数。所有的指令都是通过读操作来获得的，而且大部分指令并不进行存储器写操作。附录 B 的图 B.27 给出了一个包含了 10% 的 store 指令和 26% 的 load 指令的 MIPS 程序所产生的写操作，占全部存储器通信量的 $10\% / (100\% + 26\% + 10\%)$ ，即大约 7%；而占数据 Cache 通信量的 $10\% / (26\% + 10\%)$ ，即大约 28%。这样，加快经常性事件的速度就意味着要优化读 Cache 的时间，尤其是对传统处理器，它们需要等待读操作完成，但却不必等待写操作完成就可以进行其他操作。然而，Amdahl 定律（见 1.9 节）提醒我们，高性能的设计不能忽视写操作速度对系统性能的影响。

幸运的是，提高经常性事件的速度并不困难。块可以在标志位读和比较的同时被读出，所以读块操作在获得块地址的同时就开始了。如果读命中，块中所需信息立刻被传送到处理器。如果发生缺失，这样做虽然没有得到收益——但是在桌面机和服务器中，这也没有什么坏处，只需把读到的信息丢弃即可。

对于写操作来说，这样的优化却不适用。只有标志位有效而且地址命中时，块才能被修改。因为标志位检查不能与写数据并行进行，写操作通常要比读操作花费更长的时间。另一个复杂情况是处理器必须给出要写数据的大小，通常在 1 到 8 字节之间；只有块中的这一部分可以被改变。与写操作不同，读操作可以读取比所需字节数更多的信息。

写策略通常可以用于区分 Cache 设计。通常有两种基本策略用来写 Cache：

- 写直达 (write through)：信息被同时写到 Cache 块和更低一层存储器的块中。
- 写回法 (write back)：信息只被写入 Cache 块。只有 Cache 中该块被替换出去时，信息才会被写回到主存中。

为了减少替换时块的写回频率，通常需要使用一个称为重写 (脏) 位 (dirty bit) 的特征信息位。这个特征位表明一个块是脏的 (在 Cache 中被修改过)，还是干净的 (在 Cache 中没有被修改)。如果它是干净的，当发生缺失需替换该块时，由于下层存储器与 Cache 包含有相同的信息，则该块不必写回。

写回法和写直达法各有自己的优点。在写回法中，写操作和 Cache 存储器的速度是一致的，而且对同一块中的多次写操作仅仅需要对下层存储器一次操作。因此，写回法需要较小的存储带宽，

这种方法在多处理器的服务器系统中更具有吸引力。相对于写直达法,写回法对其他存储器层次和存储总线的使用较少,节省了功耗,因而非常适用于嵌入式应用程序。

写直达法比写回法更易实现。由于Cache总是干净的,故读操作发生缺失时不像写回法那样会导致下一级存储的写操作。写直达法的另一个优点是,下一级存储有最新的当前数据副本,简化了数据一致性。数据一致性对处理器和I/O均很重要,在第4章和第6章中已有所描述。多级Cache使得写直达法对上级Cache更具可行性,因为它仅仅需要对下一级进行写操作,而不是对主存通路上的所有存储层次都这样做。

正如我们将会看到的,多处理器和I/O是多变的:它们需要Cache利用写回法来减少访问的通信量,又希望利用写直达法来保证存储器层次结构中Cache和低层存储器数据的一致性。

在写直达法中,如果处理器的操作必须等待写操作完成才能进行时,则处理器称为是写停顿的。通常用来减少写停顿的优化策略是写缓存技术,它允许处理器把数据写入到缓冲区之后立刻继续工作,从而使处理器执行时间和存储器更新并行进行。我们将会看到,即使利用写缓存技术,写停顿仍然可能发生。

因为写操作时并不需要数据,因而在写缺失时通常采用以下两种策略:

- **写分配:** 在发生写缺失时,内存的块被读到Cache中,然后执行上个写命中时的操作。这与读缺失类似。
- **不按写分配:** 仅修改低层存储器的该块,而不将该块取到Cache中,因而写缺失不影响Cache内容。

在不按写分配中,直到程序要读一个块时,该块才被存取到Cache中;而在写分配中,只有被写的块会保存在Cache中。让我们看以下例子。

例题 假设有一个全相联映射的Cache,采用写回策略,刚开始时Cache为空。有下面5个存储器操作(方括号中为地址):

```
Write Mem[100];  
WriteMem[100];  
Read Mem[200];  
WriteMem[200];  
WriteMem[100].
```

分别求写分配和不按写分配情况下的命中次数、缺失次数。

解答: 不按写分配的情况:地址为100号的单元不在Cache中,而Cache对写操作不产生分配,所以前面2个写操作是缺失的。200号单元也不在Cache中,读操作缺失,但是该单元所在的块已被取到Cache中,故下一条读操作命中,最后一条写100号单元操作仍然发生缺失。不按写分配的最终结果是4次缺失和1次命中。

写分配的情况:对100号单元和200号单元第一次操作都发生缺失,其所在的块同时被读到Cache中,则剩下的操作都是命中的。故写分配总共有2次缺失和3次命中。

虽然这两种写缺失策略都可以应用到写回法或写直达法中,但是写回法Cache通常利用写分配(期望接下来对该块的写操作能够在Cache中命中)策略,而写直达法经常与不按写分配(因为接下来对该块的写操作仍必须写入到下一层存储器中,所以采用写分配并不能获得额外的收益)策略配合使用。

一个例子：Opteron 数据 Cache

为了说明这些思想，图 C.5 给出了 AMD 的 Opteron 处理器中数据 Cache 的组织结构。该数据 Cache 容量为 65 536 字节 (64 KB)，块大小为 64 字节，使用 2 路组相联映射方式，LRU 替换策略，写回法，在写缺失时使用写分配策略。

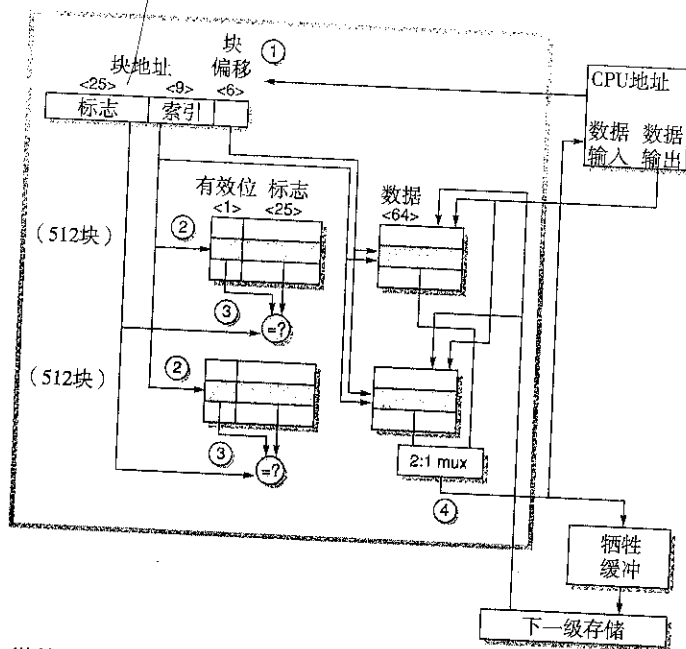


图 C.5 Opteron 微处理器中数据 Cache 的组织结构。64 KB 的 Cache 使用 64 字节块，采用 2 路组相联映射，利用 9 位索引可以对 512 个组进行选择。读命中的四个步骤以圆圈起来的数字显示，按照发生的次序标明了组织顺序。块内地址偏移的 3 位放到索引字段中，提供给 RAM 地址以选择正确的 8 个字节。这样，Cache 被划分成 2 路，每路 4096 个长度为 64 位的字，每路包括 $512/2 = 256$ 个组。尽管未在此例中标注，当 Cache 缺失时，从存储器调入一行来给 Cache 加载数据。由于是物理地址而不是虚拟地址，处理器发生的地址长度为 40 位。图 C.23 给出了 Opteron 处理器在 Cache 访问时，虚拟地址到物理地址的映射关系

在图 C.5 中，可以根据标识的步骤顺序来跟踪一次 Cache 命中过程 (四个步骤由圆圈中的数字表示)。正如在 C.5 节中看到的，Opteron 微处理器在 Cache 中使用 48 位的虚拟地址来做标志比较，同时将其转化为 40 位的物理地址。

Opteron 没有用上全部 64 位虚地址的原因，是它的设计者们认为目前还不会用到如此大的虚拟地址空间，另外，减少地址位数同时也简化了 Opteron 的虚地址映射设计。Opteron 的设计者计划在以后的微处理器产品中增加虚拟地址空间。

进入 Cache 的物理地址被分为两部分：34 位块地址和 6 位块内地址偏移 ($64 = 2^6$, $34 + 6 = 40$)。块地址被进一步分为地址标志字段和 Cache 索引字段。步骤 1 给出了这个划分。

Cache 索引字段用于确定哪一个标志将被检验，将该标志与处理器发出的访存地址中的标志进行比较，以判断要访问的块是否在 Cache 中。索引字段的宽度与 Cache 的大小、块大小及组相联度有关。Opteron 的 Cache 组相联度为 2，索引字段宽度的计算如下：

$$2^{\text{Index}} = \frac{\text{Cache 大小}}{\text{块大小} \times \text{组相联度}} = \frac{65\,536}{64 \times 2} = 512 = 2^9$$

因此索引字段宽度为9，而标志字段宽度为 $34 - 9 = 25$ 位。我们知道为得到正确的块，需要比较索引字段，但64字节相对于处理器的一次处理能力显得太大了，因此，又将Cache存储器的数据部分划分成若干个8字节，即64位Opteron处理器的一次处理长度。因此，用索引字段的9位来选择正确的Cache块，再用块内偏移地址字段中的3位来确定需要的8字节。图C.5中的步骤2表示的是由索引选择标志的过程。

从Cache中读出2个标志之后，它们被用来同从处理器发来的块地址中的标志字段部分进行比较。这是图中的第3步。为了保证标志字段中包含有效的信息，必须要设置有效位，否则比较的结果将被忽略。

假定有一个标志字段匹配，则最后一步是通知处理器根据2选1多路选择器有效输出从Cache中加载的正确数据。Opteron允许在2个时钟周期内完成这四个步骤。如果在随后两个时钟周期内的指令要使用这些数据，那么它们需要等待加载操作的结果。

在Opteron中，写操作的处理比读操作的处理过程更为复杂，这同任何Cache的情况都是一致的。如果要写的字已经在Cache中，前三个步骤都是一样的。由于Opteron是乱序执行的，只有等到指令提交并且Cache标志检验结果是命中时，数据才被写到Cache中。

到目前为止，我们假定的都是Cache命中时的情况，如果缺失会发生什么情况呢？如果读操作缺失，Cache就向处理器发出一个向其表明当前所需数据不可用的信号，然后从层次结构的下一级中读出64个字节。对于块中的前8个字节，延迟是7个时钟周期，对于其余的块，每8个字节需2个时钟周期。因为数据Cache是组相联映射的，还涉及到替换的选择问题。Opteron使用LRU选择最近最少使用的那个块，因此，每次访问需要更新LRU位。替换一个块意味着要更新数据、地址标志字段、有效位以及LRU位。

Opteron采用写回法，一个被替换掉的块可能已经被修改，因此不能简单地将其丢弃。Opteron使用1位重写位记录该块是否曾经被修改。如果已经被修改，则将该块的数据和地址送至牺牲缓存（该结构与其他计算机中的写缓存相似）。Opteron中的牺牲缓存由8个牺牲块组成，当它将替换出来的牺牲块写回低一级存储器时，可与其他Cache操作并行执行。如果牺牲缓存已满，Cache就必须停下来等待。

Opteron采用写分配法，为读缺失或写缺失都分配一个Cache块，故写缺失与读缺失操作相类似。

我们已经看到了数据Cache是如何工作的，但是它并不能支持处理器所有的存储器访问请求：处理器还需要处理指令。虽然可以采用一体Cache同时提供指令和数据，但这种方法会使一体Cache成为性能瓶颈。例如，当执行一条load指令或store指令时，指令流水的处理器会同时请求一个数据字和一个指令字。因此一体Cache会在load或store操作时引起结构冒险，造成停顿。解决这个问题的一个简单方法是把Cache划分成指令Cache和数据Cache。分离的Cache技术已被应用到现在的处理器中，包括Opteron，它包含一个64 KB的指令Cache和一个64 KB的数据Cache。

处理器知道它发射的是指令地址还是数据地址，可以为两者设置独立的端口，从而使处理器和存储器层次结构间的带宽得以加倍。指令Cache和数据Cache的独立性也为各自进行独立的优化提供了可能：不同的容量、块大小以及相联度可以获得更好的性能（与Opteron的指令Cache和数据Cache相对应，术语一体或混合表示既包含指令又包含数据Cache）。

图C.6说明指令Cache比数据Cache有更低的缺失率。通过指令Cache和数据Cache独立设置，能够消除因指令和数据冲突而引起的缺失，但是独立性也限定了两种Cache各自的大小。究竟哪一个因素对缺失率影响更大呢？分离的指令和数据Cache与一体Cache之间合理的比较应该在Cache总容量相同的情况下进行。例如，一个分离的16 KB的指令Cache和16 KB的数据Cache应该同32 KB的一体Cache相比较。计算分离Cache的平均缺失率必须要知道对指令Cache和数据Cache

各自的访问频率。图 B.27 表明采用分离 Cache 时, 对指令 Cache 的访问频率是 100%(100% + 26% + 10%)即 74%, 对数据 Cache 的访问频率是(26% + 10%)/(100% + 26% + 10%)即 26%。正如我们将会看到的, 分离的 Cache 对性能的影响超出了缺失率变化所带来的影响。

容量	指令 Cache	数据 Cache	一体 Cache
8 KB	8.16	44.0	63.0
16 KB	3.82	40.9	51.0
32 KB	1.36	38.4	43.3
64 KB	0.61	36.9	39.4
128 KB	0.30	35.3	36.2
256 KB	0.02	32.6	32.9

图 C.6 不同容量指令 Cache、数据 Cache 以及一体 Cache 的每千条指令缺失率。指令访问频率大约是 74%。数据 Cache 块大小是 64 字节, 采用 2 路组相联映射方式。结果是通过在与图 C.4 相同的系统上运行相同的测试程序获得的

C.2 Cache 性能

由于指令数是独立于硬件的, 因此可以用来评价处理器的性能。但是间接的性能评价方式会带来很多问题。与此类间接方式对应的存储器层次结构性能的评价方法主要是以缺失率为评价标准, 因为它是独立于硬件速度的。我们将会看到, 缺失率同指令数一样, 也会产生误导。对存储器层次结构更好的评价方法采用了平均存储器访问时间:

$$\text{平均存储器访问时间} = \text{命中时间} + \text{缺失率} \times \text{缺失代价}$$

这里的命中时间是 Cache 命中时所需要的时间; 另外两个术语在前面已经看到过。平均存储器访问时间的单位可以采用绝对时间——如一次命中需 0.25~1.0 ns ——或是用处理器等待存储器的时钟周期数——如缺失代价用 150~200 个时钟周期表示。注意, 平均存储器访问时间仍然是一个间接的性能评价方法; 虽然它比用缺失率更好, 但它不能代替执行时间。

这个公式能够帮助我们在分立 Cache 和一体 Cache 间做出抉择。

例题 一个 16 KB 指令 Cache 加一个 16 KB 数据 Cache 与一个 32 KB 的一体 Cache 相比较, 哪一个具有更低的缺失率? 假设 36% 的存储器访问是数据访问, 使用图 C.6 中的缺失率来计算正确的结果。假定 Cache 命中需要 1 个时钟周期, 缺失代价是 100 个时钟周期, 在一体 Cache 中, load 或 store 命中额外需要一个时钟周期, 因为只有一个 Cache 端口来满足这两个同时发生的请求。使用第 2 章提到的流水线术语, 即一体 Cache 导致结构冒险。每种情况下的平均存储器访问时间是多少呢? 假定是带有写缓存的写直达 Cache, 而且写缓存的停顿时间可以忽略不计。

解答: 先将每 1000 条指令的缺失次数转化为缺失率:

$$\text{缺失率} = \frac{\frac{\text{缺失次数}}{1000 \text{ 条指令}}}{\frac{\text{内存访问次数}}{\text{指令数}}}$$

一条指令只进行一次取指令的访存操作, 故指令的缺失率为

$$\text{缺失率}_{16 \text{ KB 指令 Cache}} = \frac{3.82/1000}{1.00} = 0.004$$

由题目的假设可知 36% 的指令是数据操作指令, 故数据缺失率为

$$\text{缺失率}_{16 \text{ KB 数据 Cache}} = \frac{40.9/1000}{0.36} = 0.114$$

一体 Cache 的缺失率要包括指令和数据的缺失率:

$$\text{缺失率}_{32 \text{ KB 一体 Cache}} = \frac{43.3/1000}{1.00 + 0.36} = 0.0318$$

根据以上情况, 74% 的存储器访问和指令相关。因此, 分立 Cache 的全局缺失率为

$$(74\% \times 0.004) + (26\% \times 0.114) = 0.0326$$

因此, 32 KB 的一体 Cache 比 2 个 16 KB 的 Cache 有相对较低的缺失率。

平均存储器访问时间公式可以分为指令和数据的访问:

$$\begin{aligned} \text{平均存储器访问时间} = & \text{指令所占比例} \times (\text{命中时间} + \text{指令缺失率} \times \text{缺失代价}) + \\ & \text{数据所占比例} \times (\text{命中时间} + \text{指令缺失率} \times \text{缺失代价}) \end{aligned}$$

所以, 两种 Cache 结构的平均存储器访问时间如下:

$$\begin{aligned} \text{平均存储器访问时间}_{\text{分立 Cache}} = & 74\% \times (1 + 0.004 \times 200) + 26\% \times (1 + 0.114 \times 200) = \\ & (74\% \times 1.80) + (26\% \times 23.80) = 1.332 + 6.118 = 7.52 \end{aligned}$$

$$\begin{aligned} \text{平均存储器访问时间}_{\text{一体 Cache}} = & 74\% \times (1 + 0.0318 \times 200) + 26\% \times (1 + 1 + 0.0318 \times 200) = \\ & (74\% \times 7.36) + (26\% \times 8.36) = 5.446 + 2.174 = 7.62 \end{aligned}$$

本例中由于分立 Cache 在每个时钟周期提供两个存储器端口, 因此, 可以有效避免结构冒险——比只有一个端口的一体 Cache 有更好的平均存储器访问时间, 尽管它的缺失率更高一些。

平均存储器访问时间和处理器性能

由上面的讨论, 读者可能会问: 是否可以由 Cache 缺失引起的平均存储器访问时间来预测处理器性能呢?

首先, 还有其他原因可以引起停顿, 例如由于 I/O 设备使用存储器引起的竞争。在评价不同 Cache 方案时, 设计者常常假定所有的存储器停顿都是由于 Cache 缺失引起的, 这是因为存储器层次结构相对于其他因素而言, 对处理器停顿影响更大。在这里, 我们也使用这个简化假定, 但在最后的性能评价时, 将由各种因素引起的存储器停顿都考虑在内是至关重要的!

另外, 这个问题的答案也与处理器的类型有关。如果处理器是顺序执行的 (见第 3 章), 处理器在发生缺失期间的停顿时间以及存储器停顿时间都与平均存储器访问时间密切相关。现在不妨假设处理器是顺序执行的, 但在下一小节中我们将重新回到对乱序执行处理器的讨论。

如前一节所述, CPU 时间可用下面的公式计算:

$$\text{CPU 时间} = (\text{CPU 执行时钟周期数} + \text{存储器停顿时钟周期数}) \times \text{时钟周期}$$

这个 CPU 时间公式产生了这样一个问题, 即 Cache 命中的时钟周期数应该被认为是 CPU 执行时钟周期数的一部分呢? 还是存储器停顿时钟周期数的一部分呢? 虽然两种观点均有道理, 但是一般的观点还是把命中时钟周期数包含在 CPU 执行时钟周期数中。

现在我们研究 Cache 对处理器性能的影响。

例题 我们使用一个顺序执行的计算机作为第一个例子。假定 Cache 缺失代价是 200 个时钟周期, 而且所有指令都用 1.0 个时钟周期完成 (忽略存储器停顿)。假定平均缺失率是 2%, 平均每条指令要访问存储器 1.5 次, 每 1000 条指令平均 Cache 缺失次数为 30。Cache 的行为会对处理器性能产生怎样的影响呢? 分别用缺失率和每条指令缺失次数计算 Cache 对计算机性能的影响。

解答: CPU时间 = 指令数 \times $\left(\text{指令执行时钟周期数} + \frac{\text{存储器停顿时钟周期}}{\text{指令数}} \right) \times \text{时钟周期时间}$

$$\begin{aligned} \text{CPU 时间}_{\text{Cache}} &= \text{指令数} \times (1.0 + (30/1000 \times 200)) \times \text{时钟周期时间} \\ &= \text{指令数} \times 7.00 \times \text{时钟周期时间} \end{aligned}$$

用缺失率计算的性能为

$$\text{CPU 时间} = \text{指令数} \times \left(\text{指令执行时钟周期数} + \text{缺失率} \times \frac{\text{存储器访问次数}}{\text{指令数}} \times \text{缺失代价} \right) \times \text{时钟周期时间}$$

$$\begin{aligned} \text{CPU 时间}_{\text{Cache}} &= \text{执行指令数} \times (1.0 + (1.5 \times 2\% \times 200)) \times \text{时钟周期时间} \\ &= \text{执行指令数} \times 7.00 \times \text{时钟周期时间} \end{aligned}$$

无论是否具有 Cache, 系统的时钟周期时间和指令数是相同的, CPI (每条指令执行时钟周期数) 从 1.0 (理想情况下, 不考虑 Cache 缺失) 增加到 7.00 (考虑 Cache 缺失), 从而导致 CPU 时间增加 7 倍。如果不采用存储器层次结构, CPI 将会增加到 $1.0 + 200 \times 1.5 = 301$ 倍, 这几乎是带有 Cache 的系统的 40 倍。

正如本例中所指出的那样, Cache 的行为将会对处理器的性能产生极大的影响。即 Cache 缺失会对具有较低 CPI 和较快时钟频率的处理器有双重影响:

1. CPI_{执行} 越低, 一定的 Cache 缺失时钟数对 CPU 时间的相对影响越大。
2. 在计算 CPI 时 (考虑 Cache 缺失情况), Cache 缺失代价用缺失时花费的处理器时钟周期数来度量。因此, 如果两台计算机的存储器层次结构相同, 那么对于时钟频率较高的处理器, 每次缺失会花费较多的时钟周期数, 因此 CPI 花费在存储器部分的周期较多。

Cache 对于具有低 CPI 和高时钟频率处理器的性能影响更为重要, 因此, 在评价这种计算机系统性能时, 如果忽略 Cache 的行为也就更为危险。Amdahl 定律再次得到了验证!

虽然追求最小平均存储器访问时间是一个合理的目标, 且本附录的大部分将会用到它, 但要注意我们的最终目标是降低处理器执行时间。下面的例子将分析两者的区别。

例题 两种不同的 Cache 组织形式究竟对处理器的性能有什么样的影响呢? 假定 Cache 在理想状态时, CPI 为 1.6, 时钟周期时间为 0.35 ns, 平均每条指令需要访问存储器 1.4 次, 而且两个 Cache 的容量都是 128 KB, 块容量都为 64 字节。一个 Cache 采用直接映射方式, 另一个采用 2 路组相联映射方式。图 C.5 说明了对于组相联的 Cache, 必须增加一个多路选择器来依据标志匹配从某一组中选择出所需的块。因为处理器速度是同 Cache 命中时的速度直接相关的, 所以假定处理器的时钟周期时间必须扩展 1.35 倍才能与组相联 Cache 的多路选择时间相适应。第一个近似假设 Cache 的缺失代价对于每种 Cache 组织形式都是 65 ns (实际上, 它必定为时钟周期的整数倍)。首先计算平均存储器访问时间, 然后计算处理器的性能。假定命中时间是 1 个时钟周期, 并假定直接映射、容量为 128 KB 的 Cache 的缺失率为 2.1%, 而同样大小的采用 2 路组相联映射的 Cache 的缺失率为 1.9%。

解答: 平均存储器访问时间为

$$\text{平均存储器访问时间} = \text{命中时间} + \text{缺失率} \times \text{缺失代价}$$

因此, 对于每一种组织形式来说,

$$\text{平均存储器访问时间}_{1\text{路}} = 0.35 + (0.21 \times 65) = 1.72 \text{ ns}$$

$$\text{平均存储器访问时间}_{2\text{路}} = 0.35 \times 1.35 + (0.19 \times 65) = 1.71 \text{ ns}$$

所以2路组相联的存储器访问时间性能更好。

处理器性能为

$$\begin{aligned}\text{CPU时间} &= \text{执行指令数} \times \left(\text{指令执行周期数} + \frac{\text{缺失次数}}{\text{指令数}} \times \text{缺失代价} \right) \times \text{时钟周期时间} \\ &= \text{执行指令数} \times \left[(\text{指令执行周期数} \times \text{时钟周期时间}) \right. \\ &\quad \left. + \left(\text{缺失率} \times \frac{\text{存储器访问次数}}{\text{指令数}} \times \text{缺失代价} \times \text{时钟周期时间} \right) \right]\end{aligned}$$

用65 ns替换乘式(缺失代价×时钟周期时间),则采用不同Cache组织形式的处理器性能为

$$\text{CPU时间}_{1\text{路}} = \text{IC} \times (1.6 \times 0.35 + (0.021 \times 1.4 \times 65)) = 2.47 \times \text{指令执行数}$$

$$\text{CPU时间}_{2\text{路}} = \text{IC} \times (1.6 \times 0.35 \times 1.35 + (0.019 \times 1.4 \times 65)) = 2.49 \times \text{指令执行数}$$

相对性能为

$$\frac{\text{CPU时间}_{2\text{路}}}{\text{CPU时间}_{1\text{路}}} = \frac{2.49 \times \text{指令数}}{2.47 \times \text{指令数}} = \frac{2.49}{2.47} = 1.01$$

与比较平均存储器访问时间时所得到的结果相反,直接映射Cache的平均性能稍微好一些,这是因为在2路组相联情况下,尽管它的缺失率低一些,但是所有指令时钟周期都延长了。由于CPU时间是我们的基本评估标准,而且直接映射更容易实现,所以在这个例子中直接映射Cache是更好的选择。

缺失代价和乱序执行处理器

对于一个乱序执行处理器,如何定义它的缺失代价呢?是存储器缺失的全部延迟时间?还是处理器必须停顿的时间或不重叠的延迟时间?这个问题在数据缺失的处理完成之前采用停顿处理的处理器中是不存在的。

我们可以重新定义存储器停顿时间,以得到重叠延迟时间表示的缺失代价定义:

$$\frac{\text{存储器停顿时钟周期}}{\text{指令数}} = \frac{\text{缺失次数}}{\text{指令数}} \times (\text{全部缺失延迟} - \text{重叠缺失延迟})$$

与此类似,由于一些乱序执行处理器延长了命中时间,性能公式的一部分内容应由全部缺失延迟减去重叠缺失延迟的差值代替。考虑到乱序执行处理器中的存储器资源竞争,该公式可以进一步扩展,将全部缺失延迟划分成没有竞争时的延迟和有竞争时的延迟。我们只关心缺失延迟。

我们需要回答下面两个问题:

- **存储器时延长度:** 在乱序处理器中,如何确定存储器操作的开始和结束时刻?
- **延迟重叠长度:** 如何确定处理器中重叠的开始时刻(或者说,什么时刻可以说存储器操作引起处理器停顿)?

由于乱序执行处理器的复杂性,目前这还没有一个准确的定义。

因为在流水线退出阶段只有已经提交的操作是可见的,因此,如果在一个周期内处理器不能完成最大可能数目的指令,则称处理器在这个时钟周期内停顿。这个停顿是由第一条不能被完成的指令引起的。该定义不像它看上去那样浅显。比如,针对某一类停顿时间采用的优化措施可能会引起其他类型的停顿,因此,它不可能总会提高处理器执行时间。

时延开始时间的确定有下面三种方式：从存储器指令进入指令窗口的时刻算起；或者从地址产生的时刻算起；或是从指令实际被送到存储器系统的时刻算起。只要采用一致的标准，任何一种方式都是适用的。

例题 重新计算上一个例子，这次假设通过加长处理器的时钟周期来支持乱序执行，仍采用直接映射 Cache。假设缺失代价为 65 ns，其中 30% 是重叠的，也就是说，平均 CPU 存储器停顿时 间现在为 45.5 ns。

解答：乱序 (OOO) 处理器的平均存储器访问时间是

$$\text{平均存储器访问时间}_{\text{乱序}} = 0.35 \times 1.35 + (0.021 \times 45.5) = 1.43 \text{ ns}$$

OOO Cache 的性能为

$$\begin{aligned} \text{CPU 时间}_{\text{乱序}} &= \text{执行指令数} \times (1.6 \times 0.35 \times 1.35 + (0.021 \times 1.4 \times 45.5)) \\ &= 2.09 \times \text{执行指令数} \end{aligned}$$

由此可见，尽管时钟周期较慢，直接映射的 Cache 缺失率较高，但乱序执行处理器如果能重叠 30% 的缺失代价，速度也能快一点。

总之，对于乱序执行处理器，尽管存储器停顿的定义和评价问题相对复杂，但仍需关注它，因为它对性能有很大的影响。复杂的原因在于乱序执行处理器能在不降低性能的情况下容忍 Cache 缺失。因此，设计者往往使用乱序执行处理器和存储器的模拟器来评估存储层次结构的折中，以提高平均存储时延来改善程序性能。

图 C.7 中列出了本章中有关 Cache 的计算公式，以供读者参考。

$$2^{\text{index}} = \frac{\text{Cache 大小}}{\text{块大小} \times \text{组相联度}}$$

$$\text{CPU 执行时间} = (\text{CPU 时钟周期数} + \text{存储器停顿周期数}) \times \text{时钟周期时间}$$

$$\text{存储器停顿周期数} = \text{缺失次数} \times \text{缺失代价}$$

$$\text{存储器停顿周期数} = \text{执行指令数} \times \frac{\text{缺失次数}}{\text{指令数}} \times \text{缺失代价}$$

$$\frac{\text{缺失次数}}{\text{指令数}} = \text{缺失率} \times \frac{\text{存储器访问次数}}{\text{指令数}}$$

$$\text{平均存储器访问时间} = \text{命中时间} + \text{缺失率} \times \text{缺失代价}$$

$$\text{CPU 执行时间} = \text{指令数} \times \left(\text{指令执行时钟周期数} + \frac{\text{存储器停顿时钟周期}}{\text{指令数}} \right) \times \text{时钟周期时间}$$

$$\text{CPU 执行时间} = \text{指令数} \times \left(\text{指令执行时钟周期数} + \frac{\text{缺失次数}}{\text{指令数}} \times \text{缺失代价} \right) \times \text{时钟周期时间}$$

$$\text{CPU 执行时间} = \text{指令数} \times \left(\text{指令执行时钟周期数} + \text{缺失率} \times \frac{\text{存储器访问次数}}{\text{指令数}} \times \text{缺失代价} \right) \times \text{时钟周期时间}$$

$$\frac{\text{存储器停顿时钟周期}}{\text{指令数}} = \frac{\text{缺失次数}}{\text{指令数}} \times (\text{全部缺失延迟} - \text{重叠缺失延迟})$$

$$\text{平均存储器访问时间} = \text{命中时间}_{L1} + \text{缺失率}_{L1} \times (\text{命中时间}_{L2} + \text{缺失率}_{L2} \times \text{缺失代价}_{L2})$$

$$\frac{\text{存储器停顿时钟周期}}{\text{指令数}} = \frac{\text{缺失次数}_{L1}}{\text{指令数}} \times \text{命中时间}_{L2} + \frac{\text{缺失次数}_{L2}}{\text{指令数}} \times \text{缺失代价}_{L2}$$

图 C.7 本章中性能计算公式的总结。第一个公式用来计算 Cache 索引字段长度，余下的公式用来计算性能。考虑完备性，加了最后两个公式，用于多级 Cache 的计算，这两个公式将在下一节的开始部分加以解释说明

C.3 六种基本的 Cache 优化

平均存储器访问时间公式提供了一个用于优化 Cache 的性能或功耗的框架:

$$\text{平均存储器访问时间} = \text{命中时间} + \text{缺失率} \times \text{缺失代价}$$

因此,我们把6种Cache的优化策略归结为以下三类:

- 降低缺失率: 增大块容量, 增大 Cache 容量, 增加相联度。
- 减少缺失代价: 多级 Cache, 读缺失优先于写。
- 减少 Cache 命中时间: 避免索引 Cache 时的地址转换。

图 C.17 将对这6种技术的实现复杂度和性能改善进行总结。

提高 Cache 性能的经典方法是降低缺失率。这里将介绍3种技术。首先, 为了对引起缺失原因有更好的认识, 首先来研究一个模型, 它将所有缺失归结为三种类型:

- **强制缺失**: 对一个块的第一次访问一定不在 Cache 中, 所以该块必须被调入到 Cache 中。这也称为**冷启动缺失**或是**首次访问缺失**。
- **容量缺失**: 如果 Cache 容纳不了一个程序执行所需要的所有块, 将会发生容量缺失(还会发生强制缺失), 某些块将被丢弃, 随后再被调入。
- **冲突缺失**: 如果块替换策略是组相联或是直接映射的, 并且有太多的块被映射到同一组中, 则某一个块被放弃, 之后重新再调入, 这时就发生了**冲突缺失**(另外还有强制缺失和容量缺失)。这也称为**碰撞缺失**或**干涉缺失**。也即在全相联 Cache 中命中, 却在 n 路组相联 Cache 中缺失, 这是由于对某些组块数的要求越过了 n 。

我们将以上三类缺失称为3C缺失, 第4章加入了第4个C, 即一致(coherency)缺失, 由于 Cache 刷新而保持在多处理器中的多 Cache 一致性, 这里我们不做考虑。

图 C.8 所示为依据3C缺失类型划分的 Cache 的相对缺失频率。强制缺失发生在无限大的 Cache 中; 容量缺失发生在全相联 Cache 中; 冲突缺失在全相联、8路组相联和4路组相联等 Cache 中都出现。图 C.9 以图形的形式给出了同样的数据。上部的图形表示绝对缺失率; 底部的图形画出了所有类型缺失的缺失率百分比, 它是 Cache 大小的函数。

组相联度的下降会引起冲突缺失, 为显示相联度的效果, 我们将冲突缺失划分为以下四组, 并给出其计算方法:

- **8路**: 从全相联(没有冲突)到8路相联而引起的冲突缺失。
- **4路**: 从8路相联到4路相联而引起的冲突缺失。
- **2路**: 从4路相联到2路相联而引起的冲突缺失。
- **1路**: 从2路相联到1路相联(直接映射)而引起的冲突缺失。

正如我们在图中所见, 程序 SPEC2000 的强制缺失率很小, 这一点同许多运行时间长的程序一致。

定义了3C缺失率后, 计算机设计者根据这几种缺失情况可以做些什么呢? 从理论上来讲, 冲突缺失是最容易解决的: 全相联映射方案避免了所有的冲突缺失。然而, 采用全相联所需的硬件价格昂贵, 而且可能会使处理器的时钟频率变慢(见上面的例子), 这将会导致系统的总性能降低。

Cache 容量 (KB)	相联度	总缺 失率	缺失率因子 (相对百分比) (总和为总缺失率的 100%)					
			强制缺失		容量缺失		冲突缺失	
4	1 路	0.098	0.0001	0.1%	0.070	72%	0.027	28%
4	2 路	0.076	0.0001	0.1%	0.070	93%	0.005	7%
4	4 路	0.071	0.0001	0.1%	0.070	99%	0.001	1%
4	8 路	0.071	0.0001	0.1%	0.070	100%	0.000	0%
8	1 路	0.068	0.0001	0.1%	0.044	65%	0.024	35%
8	2 路	0.049	0.0001	0.1%	0.044	90%	0.005	10%
8	4 路	0.044	0.0001	0.1%	0.044	99%	0.000	1%
8	8 路	0.044	0.0001	0.1%	0.044	100%	0.000	0%
16	1 路	0.049	0.0001	0.1%	0.040	82%	0.009	17%
16	2 路	0.041	0.0001	0.2%	0.040	98%	0.001	2%
16	4 路	0.041	0.0001	0.2%	0.040	99%	0.000	0%
16	8 路	0.041	0.0001	0.2%	0.040	100%	0.000	0%
32	1 路	0.042	0.0001	0.2%	0.037	89%	0.005	11%
32	2 路	0.038	0.0001	0.2%	0.037	99%	0.000	0%
32	4 路	0.037	0.0001	0.2%	0.037	100%	0.000	0%
32	8 路	0.037	0.0001	0.2%	0.037	100%	0.000	0%
64	1 路	0.037	0.0001	0.2%	0.028	77%	0.008	23%
64	2 路	0.031	0.0001	0.2%	0.028	91%	0.003	9%
64	4 路	0.030	0.0001	0.2%	0.028	95%	0.001	4%
64	8 路	0.029	0.0001	0.2%	0.028	97%	0.001	2%
128	1 路	0.021	0.0001	0.3%	0.019	91%	0.002	8%
128	2 路	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128	4 路	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128	8 路	0.019	0.0001	0.3%	0.019	100%	0.000	0%
256	1 路	0.013	0.0001	0.5%	0.012	94%	0.001	6%
256	2 路	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	4 路	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	8 路	0.012	0.0001	0.5%	0.012	99%	0.000	0%
512	1 路	0.008	0.0001	0.8%	0.005	66%	0.003	33%
512	2 路	0.007	0.0001	0.9%	0.005	71%	0.002	28%
512	4 路	0.006	0.0001	1.1%	0.005	91%	0.000	8%
512	8 路	0.006	0.0001	1.1%	0.005	95%	0.000	4%

图 C.8 不同容量Cache的总缺失率以及根据缺失类型划分的各种缺失率百分比。强制缺失率与Cache大小无关；容量缺失率随容量的增长而下降；冲突缺失率随相联程度的增加而下降。图 C.9 以图形形式给出了同样的信息。注意，表中 4 KB 到 128 KB 容量Cache的数据验证了 2:1 Cache 经验规律：容量为 N 的直接映射Cache同容量为 $N/2$ 的 2 路组相联Cache有着大致相同的缺失率。当容量大于 128 KB 时，该规则不成立。强制缺失率是按照全相联Cache的缺失率计算出的。数据来源与图 C.4 相同，使用了 LRU 替换策略

对于容量缺失来说，增大Cache的容量是唯一可能的途径。如果上层的存储器比程序所需要的空间小很多，那么在层次结构两层之间传输数据就会耗费更长的时间，这就是存储器层次结构的抖动现象。由于抖动需要多次替换，因此，它使得机器的速度接近于较低层存储器的速度，而且机器的速度有可能由于缺失的代价变得更慢。

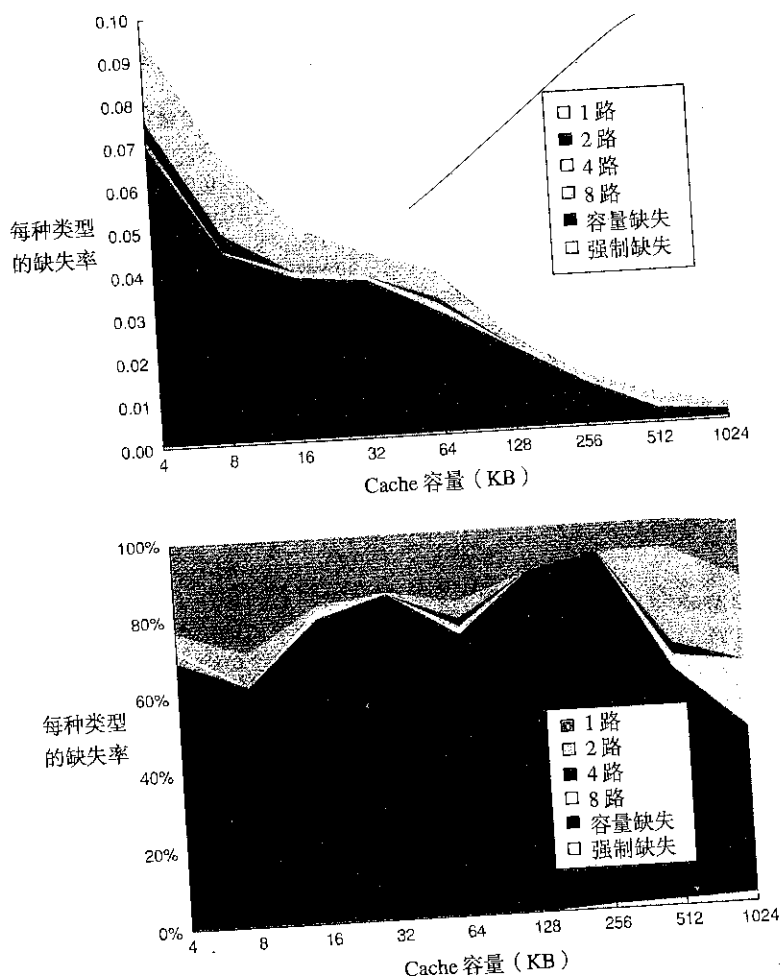


图 C.9 根据图 C.8 的数据, 按照缺失类型划分的不同容量 Cache 的总缺失率 (上图) 和缺失率分布 (下图)。上图是实际的缺失率, 而下图是按照缺失类型划分的缺失百分率 (图形上空间显示能指出额外的 Cache 容量, 能更好适合图 C.8)

另外一种改善 3C 缺失率的方法是通过增加块大小以减少强制缺失发生的次数, 但正如下面将要看到的, 块容量增大会增加其他类型的缺失。

3C 缺失率指出了引起缺失的原因, 但是这个简化模型也有它的局限性; 它虽然解释了平均行为, 但不一定能够解释单个缺失的情况。例如, 改变 Cache 容量的大小会在改变容量缺失的同时改变冲突缺失, 这是因为更大的 Cache 会把访问扩展到更多的块中。因此, 若 Cache 容量改变, 则缺失可能会从容量缺失变为冲突缺失。需要注意的是, 3C 缺失率还忽略了替换策略, 这是由于它难于建模而且一般说来也不太重要。在特殊情况下, 替换策略实际上可能会导致不正常行为, 例如, 增加组相联度会使缺失率更低, 这一点与上述 3C 缺失率模型是相互矛盾的 (有些研究者建议在存储器中使用地址跟踪来决定采用什么样的优化替换策略, 以避免因替换策略造成的缺失率升高。我们并没有采用这种方法)。

许多技术在降低缺失率的同时, 也增加了命中时的开销或者缺失代价。在采用优化技术降低 3C 缺失率的同时, 必须与提高系统整体性能的目标进行折中。第一个例子表明了这种折中观点的重要性。

第一种优化技术：增加块容量来降低缺失率

最简单降低缺失率的方法是增加块容量。图 C.10 给出了一组不同 Cache 容量情况下，缺失率和块容量之间的折中关系。增加块容量会降低强制缺失率。这是由于局部性原理有两个组成部分：时间局部性和空间局部性。增加块容量利用了空间局部性原理。

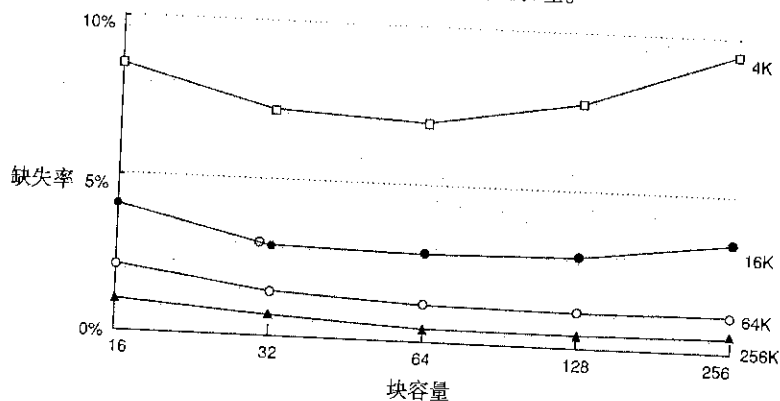


图 C.10 五种不同容量Cache的缺失率同块容量的关系。注意，如果块的大小相对于Cache的容量足够大，缺失率实际是增大的。每条线表示一个不同容量的Cache。图 C.11 给出了用来画这些线的数据。如果块容量包含在内的话，进行SPEC2000跟踪将花费很长的时间，因此，这些数据是在DECstation5000[Gee等1993]上进行SPEC92测试得到的

更大的块也增加了缺失代价。因为它减少了Cache中的块数，所以更大的块可能会导致冲突缺失，而且如果Cache容量较小时，甚至会有容量缺失。很明显，如果增加块容量而导致缺失率增加，那么增加块容量就没有任何意义了；而如果它增加了平均存储器访问时间，则同样对降低缺失率也没有什么好处；缺失代价的增加可能会抵消缺失率的降低。

例题 图 C.11 给出了图 C.10 中的实际缺失率。假定存储器系统缺失时系统开销为80个时钟周期，然后每2个时钟周期传送16个字节。因此，它能用82个时钟周期传送16个字节，用84个时钟周期传达32个字节，依此类推。对于图 C.11 中的不同Cache容量来说，块容量为多少才能使平均存储器访问时间最小呢？

块容量	Cache 容量			
	4 KB	16 KB	64 KB	256 KB
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	7.00%	2.64%	1.06%	0.51%
128	7.78%	2.77%	1.02%	0.49%
256	9.51%	3.29%	1.15%	0.49%

图 C.11 在图5.16中五种不同容量Cache的实际缺失率与块容量的关系。对于4 KB Cache来说，块容量为256字节的缺失率要比块容量为32字节的缺失率要高。在这个例子中，使用块容量256字节时要想降低缺失率，Cache的容量必须是256 KB

解答：平均存储器访问时间为

$$\text{平均存储器访问时间} = \text{命中时间} + \text{缺失率} \times \text{缺失代价}$$

下面将

平均行

同时改

，则缺

于它难

，例如，

建议在存

升高。我

降低这

观点的

如果我们假定命中时间与块容量无关,且命中时间是1个时钟周期,那么当Cache容量为1 KB、块容量为16字节时,平均存储器访问时间为

$$\text{平均存储器访问时间} = 1 + (8.57\% \times 82) = 8.027 \text{ 个时钟周期}$$

当Cache容量为256 KB、块容量为256字时,平均存储器访问时间为

$$\text{平均存储器访问时间} = 1 + (0.49\% \times 112) = 1.549 \text{ 个时钟周期}$$

图C.12给出了在这两种边界情况之间的各种块容量和Cache容量的平均存储器访问时间。粗体字的项表示对于给定Cache容量的速度最快的块容量:对于4 KB Cache来说,块容量为32字节时的速度最快;对于容量更大的Cache来说,块容量为64字节时的速度最快。这些块容量实际上就是我们今天处理器Cache所通用的块容量。

块容量	缺失代价	Cache 容量			
		4 KB	16 KB	64 KB	256 KB
16	82	8.027	4.231	2.673	1.894
32	84	7.082	3.411	2.134	1.588
64	88	7.160	3.323	1.933	1.449
128	96	8.469	3.659	1.979	1.470
256	112	11.651	4.685	2.288	1.549

图C.12 图C.11中给出的五种不同容量Cache的平均存储器访问时间和块容量之间的关系。一般Cache块取32字节或64字节。每种容量Cache的最小平均存储器访问时间用粗体字给出

在所有这些技术中,Cache的设计者都在尽力使缺失率和缺失代价达到最小。块容量的选择取决于较低层存储器的时延和带宽:在高时延和高带宽存储器中,较大容量的块更有效,因为在每次缺失时,Cache可以获得更多的字节,而缺失代价只有少量的增加。相反,低时延和低带宽的存储器希望块容量小一些,因为较大的块并不能节省多少时间。例如,小容量块缺失代价的两倍可能会接近于两倍容量块的一次缺失代价。块容量小,则块数量较多,可能会减少冲突缺失。图C.10和图C.12说明了在缺失率最小和平均存储器访问时间最小的前提下,块容量大小选择的不同。

在介绍了增加块容量对强制缺失和容量缺失的影响之后,接下来再来分析增加容量和相联度来降低缺失率的情况。

第二种优化技术:增加Cache容量来降低缺失率

降低图C.8和图C.9中容量缺失率最直接的方法就是增加Cache的容量。但这样做的结果将使得Cache命中时间延长,开销增大。这种技术在片外的Cache中常常采用。

第三种优化技术:增加相联度来降低缺失率

图C.8和图C.9给出了缺失率随相联度增加而得到改善的情况。可从这些图中得到两条一般性经验规律。第一条规律是:从应用角度看,8路组相联和相同容量的全相联Cache在降低缺失率方面同样有效。因为容量缺失是以全相联的Cache来计算的,因此,在图C.8中可以比较相联度为8路的容量缺失率,从而看到它们之间的不同。

第二条规律——称为2:1 Cache经验规律,就是容量为N的直接映射Cache与容量为N/2的2路组相联的Cache有几乎相同的缺失率。正是基于此,Cache的容量一般少于128 KB。

和许多其他例子一样,改进平均存储器访问时间的一个方面将会导致在其他方面的损失。增加块容量会降低缺失率,却增加了缺失代价,增加相联度则会导致命中时间增加。因此,为了实现更

高的处理器时钟频率,需要简单的Cache设计,正如下面的例子所示,增加相联度能够降低缺失率,但会增加缺失代价。

例题 假定增加相联度将会增加时钟周期时间,增加值如下:

$$\text{时钟周期时间}_{2\text{路}} = 1.36 \times \text{时钟周期时间}_{1\text{路}}$$

$$\text{时钟周期时间}_{4\text{路}} = 1.44 \times \text{时钟周期时间}_{1\text{路}}$$

$$\text{时钟周期时间}_{8\text{路}} = 1.52 \times \text{时钟周期时间}_{1\text{路}}$$

假定命中时间是1个时钟周期,而直接映射情况下缺失代价是25个时钟周期,二级Cache不发生缺失,而且不要求缺失代价必须为时钟周期的整数倍。使用图C.8给出的缺失率,那么哪种容量的Cache才能使得下面的三个表达式都成立呢?

$$\text{平均存储器访问时间}_{8\text{路}} < \text{平均存储器访问时间}_{4\text{路}}$$

$$\text{平均存储器访问时间}_{4\text{路}} < \text{平均存储器访问时间}_{2\text{路}}$$

$$\text{平均存储器访问时间}_{2\text{路}} < \text{平均存储器访问时间}_{1\text{路}}$$

解答: 每种相联度Cache的平均存储器访问时间为

$$\begin{aligned} \text{平均存储器访问时间}_{8\text{路}} &= \text{命中时间}_{8\text{路}} + \text{缺失率}_{8\text{路}} \times \text{缺失代价}_{8\text{路}} \\ &= 1.52 + \text{缺失率}_{8\text{路}} \times 25 \end{aligned}$$

$$\text{平均存储器访问时间}_{4\text{路}} = 1.44 + \text{缺失率}_{4\text{路}} \times 25$$

$$\text{平均存储器访问时间}_{2\text{路}} = 1.36 + \text{缺失率}_{2\text{路}} \times 25$$

$$\text{平均存储器访问时间}_{1\text{路}} = 1.00 + \text{缺失率}_{1\text{路}} \times 25$$

缺失代价对于每种情况都是相同的,所以,假定为25个时钟周期。例如,4 KB直接映射Cache的平均存储器访问时间为

$$\text{平均存储器访问时间}_{1\text{路}} = 1.00 + (0.133 \times 25) = 3.44$$

对于512 KB的8路组相联Cache来说,平均存储器访问时间为

$$\text{平均存储器访问时间}_{8\text{路}} = 1.52 + (0.006 \times 25) = 1.66$$

图C.13使用这些公式以及图C.8的缺失率,给出了每种容量Cache在各种相联度情况下的平均存储器访问时间。图中的数字表明,本例中使用的公式适用于容量不超过8 KB、组相联度不超过4的Cache。当容量大于16 KB时,较大的相联度所引起的命中时间的延长就超过了缺失次数减少节省下的时间。

Cache 容量 (KB)	相联度			
	1 路	2 路	4 路	8 路B
4	3.44	3.25	3.22	3.28
8	2.69	2.58	2.55	2.62
16	2.23	2.40	2.46	2.53
32	2.06	2.30	2.37	2.45
64	1.92	2.14	2.18	2.25
128	1.52	1.84	1.92	2.00
256	1.32	1.66	1.74	1.82
512	1.20	1.55	1.59	1.66

图 C.13 本例使用图C.8中的缺失率参数得到的平均存储器访问时间。粗体字表明这个时间值比左端的要高;也就是说,较高的相联度增加了平均存储器访问时间

注意,在本例中,对程序的其余部分,没有考虑较慢的时钟频率,因此低估了直接映射 Cache 的优点。

第四种优化技术:使用多级 Cache 来降低缺失代价

减少缺失次数是传统 Cache 研究的方向,但是 Cache 性能公式说明缺失代价的改善与缺失率的改善同样重要。而且,图 5.2 表明处理器性能的增长速度比 DRAM 的要快的发展趋势,使得缺失代价的相对成本将会越来越高。

处理器和存储器之间的性能差异使得系统结构设计者开始思考这样一个问题:要想克服处理器和存储器之间不断加大的性能差距,我们应当采取哪种优化方法?是提高 Cache 的速度,还是增大 Cache 的容量?

答案是应当双管齐下,即同时优化 Cache 的性能并增大容量。通过在原来的 Cache 和存储器之间增加一级 Cache 就可以做到:第一级 Cache 可以小到足以与快速的处理器运行时钟周期时间相匹配;而第二级 Cache 能够大到足以捕捉到对内存进行的大多数访问,从而有效地减小了缺失代价。

虽然在层次结构中增加另一级 Cache 的概念很自然,也很直接,但它使得对性能的分析变得复杂化。定义第二级 Cache 并不总是很直接的。我们首先从定义两级 Cache 的平均存储器访问时间开始,用下标数字 L1 和 L2 来分别表示第一级和第二级 Cache,则最初公式为

$$\text{平均存储器访问时间} = \text{命中时间}_{L1} + \text{缺失率}_{L1} \times \text{缺失代价}_{L1}$$

其中,

$$\text{缺失代价}_{L1} = \text{命中时间}_{L2} + \text{缺失率}_{L2} \times \text{缺失代价}_{L2}$$

所以,

$$\text{平均存储器访问时间} = \text{命中时间}_{L1} + \text{缺失率}_{L1} \times (\text{命中时间}_{L2} + \text{缺失率}_{L2} \times \text{缺失代价}_{L2})$$

在这个公式中, L2 Cache 的缺失率是从第一级的剩余部分开始测得的。为了更清晰地进行表达和分析,这里对两级 Cache 系统采用了如下术语:

- **局部缺失率**: 这一级 Cache 的缺失数除以 Cache 的存储器访问总数。在一级 Cache 中,它等于缺失率_{L1}; 在二级 Cache 中,它等于缺失率_{L2}。
- **全局缺失率**: 这一级 Cache 的缺失数除以处理器产生的访存总数。使用上面的术语,一级 Cache 的全局缺失率仍为缺失率_{L1}, 二级 Cache 的全局缺失率为缺失率_{L1} × 缺失率_{L2}。

二级 Cache 的局部缺失率比较大,因为一级 Cache 存储的是最易命中的数据。这就是为什么全局缺失率是一个更有效的度量标准: 它的值体现了那些从处理器直接到存储器的存储访问所占的比例。

每条指令的缺失次数在这里要重新提出。为了不混淆局部和全局缺失率,我们通过扩展每条指令的存储器停顿周期数,来加进二级 Cache 对性能的影响。

$$\text{每条指令的平均存储器停顿周期} = \text{每条指令缺失次数}_{L1} \times \text{命中时间}_{L2} + \text{每条指令缺失次数}_{L2} \times \text{缺失代价}_{L2}$$

例题 假定在 1000 次访存中,一级 Cache 中有 40 次缺失,二级 Cache 中有 20 次缺失。两种缺失率分别为多少? 假设二级 Cache 到存储器的缺失代价为 200 个时钟周期,二级 Cache 的命

中时间为 10 个时钟周期；一级 Cache 的命中时间是 1 个时钟周期，每条指令的存储器访问次数为 1.5。求平均存储器访问时间和每条指令的平均停顿周期时间各为多少（写操作影响不计）。

解答：一级 Cache 的缺失率（局部或是全局）是 $40/1000 = 4\%$ 。二级 Cache 的局部缺失率为 $20/40 = 50\%$ 。二级 Cache 的全局缺失率为 $20/1000 = 2\%$ 。因此，

$$\begin{aligned} \text{平均存储器访问时间} &= \text{命中时间}_{L1} + \text{缺失率}_{L1} \times (\text{命中时间}_{L2} + \text{缺失率}_{L2} \times \text{缺失代价}_{L2}) \\ &= 1 + 4\% \times (10 + 50\% \times 200) = 1 + 4\% \times 110 = 5.4 \text{ 个时钟周期} \end{aligned}$$

为了得到每条指令的缺失次数，将 1000 次存储器访问除以每条指令的访存次数 1.5，得 667 条指令。因此，需要将缺失次数乘以 1.5 得到每 1000 条指令的缺失次数。每 1000 条指令一级 Cache 的缺失次数为 60 次，二级 Cache 的缺失次数为 30 次。假设数据和指令的缺失次数相等，则每条指令的平均存储器停顿周期数为

$$\begin{aligned} \text{每条指令的平均存储器停顿周期} &= \text{每条指令缺失次数}_{L1} \times \text{命中时间}_{L2} + \text{每条指令缺失次数}_{L2} \times \text{缺失代价}_{L2} \\ &= (60/1000) \times 10 + (30/1000) \times 200 \\ &= 0.060 \times 10 + 0.030 \times 200 = 6.6 \text{ 个时钟周期} \end{aligned}$$

如果从平均存储器访问时间（AMAT）中减去一级 Cache 的命中时间，再乘上每条指令的平均访存（存储器访问）次数，也能得到相同的平均每条指令的存储器停顿周期数：

$$(5.4 - 1.0) \times 1.5 = 4.4 \times 1.5 = 6.6 \text{ 个时钟周期}$$

由此可见，在多级 Cache 中，用每条指令缺失数来计算要比用缺失率进行计算更加清晰。

注意，这些公式中描述的访存操作是读操作和写操作的结合，并假定一级 Cache 采用写回法。显然，采用写直达法的一级 Cache 将会把所有的写操作发送给二级 Cache，而不仅仅是在缺失时，而且可能还要用到写缓存。

图 C.14 和图 C.15 给出了缺失率和相对执行时间随二级 Cache 的大小而变化的情况。从这些图中，我们可以得到以下两点结论。首先，当二级 Cache 要比一级 Cache 大很多时，全局 Cache 缺失率与单独的二级 Cache 缺失率是非常接近的。这与我们预想的结果是一致的。二级 Cache 局部缺失率是一级 Cache 缺失率的一个函数，它将随着一级 Cache 的变化而变化。因此，用局部缺失率来评价二级 Cache 的性能不是一个好的选择，全局 Cache 缺失率才是更有效的标准。

根据以上定义，我们可以考虑二级 Cache 的参数。两级 Cache 间首要的区别是一级 Cache 的速度影响处理器的时钟频率，而二级 Cache 的速度仅仅影响一级 Cache 的缺失代价。因此，我们可以在二级 Cache 上重新考虑那些对一级 Cache 不适用的设计方案。设计二级 Cache 要考虑两个主要问题：它会降低 CPI 中的平均存储器访问时间吗？它的开销有多大？

首先需要确定的是二级 Cache 的容量。因为一级 Cache 中的所有信息都可能会出现在二级 Cache 中，因此，二级 Cache 应该比一级 Cache 大得多。如果二级 Cache 只是稍微大一点，局部缺失率会很高。这个结论推动了二级 Cache 的大容量设计——其容量同较早计算机中的内存容量相当！

还有一个需要考虑的问题就是组相联设计对于二级 Cache 来说是否有意义。

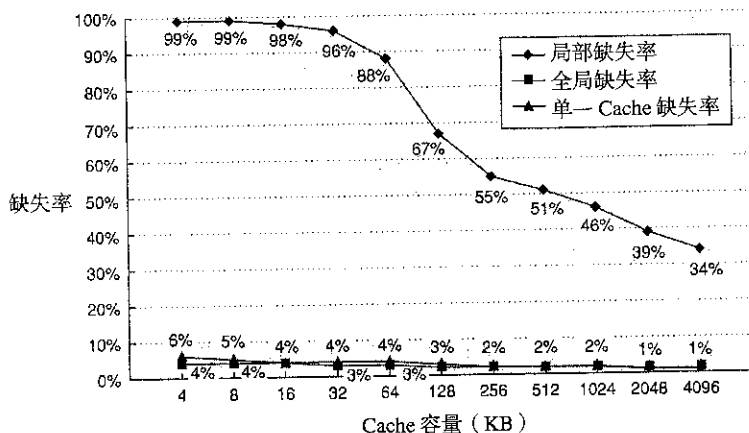


图 C.14 多级 Cache 中缺失率同 Cache 大小的关系。当二级 Cache 的容量小于两个一级 Cache 容量的和（两个一级 Cache 都为 64 KB）时，由于缺失率太高，几乎没有意义。当二级 Cache 的容量大于 256 KB 时，缺失率可以在全局缺失率的 10% 以内。单级 Cache 的缺失率同 Cache 大小的关系是相对于一个二级 Cache 的局部缺失率和全局缺失率而绘出的，该二级 Cache 的一级 Cache 容量为 32 KB。第二级一体 Cache 采用 2 路组相联和 LRU 替换策略，分别对应一级 Cache 的 64 KB 指令和 64 KB 数据 Cache。一级 Cache 也采用 2 路组相联和 Cache 替换。两级 Cache 块的大小都为 64 字节。数据来源同图 C.4

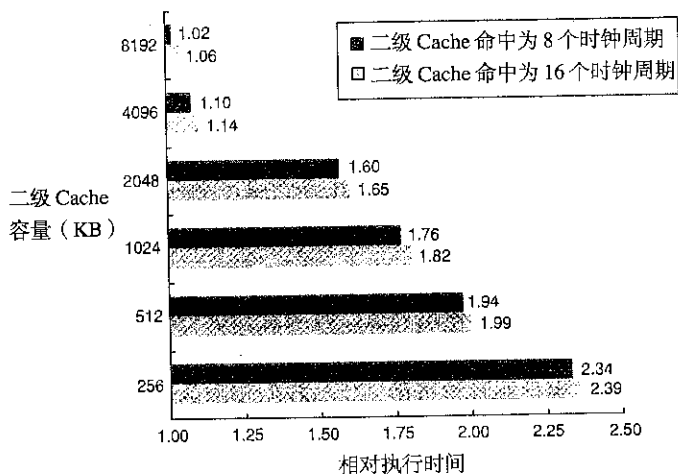


图 C.15 不同容量二级 Cache 的相对执行时间。横条表示二级 Cache 命中时不同的时钟周期数。参照执行时间 1.00 是以一个命中时有 1 个时钟周期的延迟、容量为 8192 KB 的二级 Cache 为基准的。这里的数据和图 C.14 中的数据一样，都是在 Alpha21264 上模拟运行收集到的

例题 对于下面的数据，二级 Cache 的相联度对于缺失代价的影响是什么？

- 直接映射的命中时间 $L_2 = 10$ 个时钟周期。
- 2 路组相联使命中时间增加了 0.1 个时钟周期，即命中时间为 10.1 个时钟周期。
- 直接映射的局部缺失率 $L_2 = 25\%$ 。
- 2 路组相联的局部缺失率 $L_2 = 20\%$ 。
- 缺失代价 $L_2 = 200$ 个时钟周期。

解答：二级 Cache 采用直接映射，则一级 Cache 的缺失代价为

$$\text{缺失代价}_{1\text{路}L2} = 10 + 25\% \times 200 = 60.0 \text{ 个时钟周期}$$

在二级 Cache 采用 2 路组相联后，命中开销增加了仅仅 0.1 个时钟周期，则新的一级 Cache 的缺失代价为

$$\text{缺失代价}_{2\text{路}L2} = 10.1 + 20\% \times 200 = 50.1 \text{ 个时钟周期}$$

实际上，二级 Cache 几乎总是和一级 Cache 以及处理器同步。因此，二级 Cache 的命中时间必须是时钟周期的整数倍。我们可以把二级 Cache 的命中时间缩减到 10 个周期；否则，我们可能就会舍入到 11 个周期。两种结果都比二级 Cache 采用直接映射的情况有所改善：

$$\text{缺失代价}_{2\text{路}L2} = 10 + 20\% \times 200 = 50.0 \text{ 个时钟周期}$$

$$\text{缺失代价}_{2\text{路}L2} = 11 + 20\% \times 200 = 51.0 \text{ 个时钟周期}$$

由此可见，通过减少二级 Cache 的缺失率，可以降低缺失代价。

另外一个考虑是所有一级 Cache 中的数据是否都包含在二级 Cache 中。**多级包含**是存储器层次结构的自然属性，即一级 Cache 的数据总是包含在二级 Cache 中。这是我们期望的，因为这样 I/O 和 Cache 之间（或者是在多处理器的 Cache 之间）的一致性，就能够仅仅通过检查二级 Cache 来确定。

多级包含的缺点是容量较小的一级 Cache 要用较小的块，而容量较大的二级 Cache 要用较大的块。比如，Pentium 4 的一级 Cache 块的大小为 64 字节，二级 Cache 块的大小为 128 字节。在这种情况下，为了保持这种包含关系，在二级 Cache 缺失时要做很多工作。当二级 Cache 中块被替换出去时，要使一级 Cache 中与这些块有映射关系的块无效，这将导致一级 Cache 的缺失率略微升高。为了避免这种问题，很多设计采用了各级 Cache 的块容量相等的方法。

如果二级 Cache 的容量只比一级 Cache 大一点，情况又会怎样呢？二级 Cache 的大部分只作为一级 Cache 的另一份冗余副本吗？此时可以用另一种完全相反的策略——**多级独占**：一级 Cache 中的数据都不在二级 Cache 中再现。典型的多级独占 Cache 中，一级 Cache 的缺失只会使一级 Cache 和二级 Cache 中的块发生交换，而不会用二级 Cache 中的块替换一级 Cache 中的块。这种方法防止了二级 Cache 空间浪费。例如，AMD Opteron 芯片就具有多级独占的属性，它有两个 64 KB 的一级 Cache，而二级 Cache 容量有 1 MB。

正如以上所述，尽管新手可能会分别设计一级和二级 Cache，但如果首先有了二级 Cache 的支持，那么一级 Cache 的设计就会简单得多。例如，如果下一级 Cache 采用写回法来支持重复写操作且使用了多级独占，则使用写直达可以降低风险。

所有 Cache 设计的本质都是在加快命中和减少缺失间进行折中。二级 Cache 相对于一级 Cache 有较低的命中数，因此，重点放在减少缺失次数方面，即选用大容量 Cache 以及采用降低缺失率的技术，如选择较高相联度和较大的块容量。

第五种优化技术：确定读缺失对写的优先级来降低缺失代价

这种优化是在写操作完成之前就进行读操作。我们首先来看看写缓存的复杂性。

对于一个写直达 Cache，最重要的改进是设置容量适中的写缓存。然而，因为写缓存中可能包含读缺失时所需的更新数据，这将导致存储器访问变得更复杂。

例题 请看下面的代码序列：

```
SW R3, 512(R0) ; M[512] ← R3      (cache index 0)
LW R1, 1024(R0) ; R1 ← M[1024]    (cache index 0)
LW R2, 512(R0) ; R2 ← M[512]      (cache index 0)
```

假设有一个直接映射的写直达 Cache，它把 512 和 1024 这两个地址映射到 Cache 的一个相同块中，它带有容量为 4 个字的写缓存。那么，R2 中的值总是同 R3 中的值相等吗？

解答：使用第 2 章的术语，这是一个存储器中 RAW 的数据冒险。我们通过跟踪一次 Cache 访问来看一下出现危险的可能性。R3 中的数据在执行存操作之后进入到写缓存中。接下来的 load 操作由于使用了相同的 Cache 索引，因而导致缺失。第二个 load 指令想把位置 512 处的值写到寄存器 R2 中；这也导致了一次缺失。如果写缓存还没有完成 512 处的写操作，则对 512 的读操作就把原先的错误值调入到 Cache 块中去，接着调入到 R2 中。如果没有正确的防范措施，R3 的值与 R2 的值将不等！

解决这个问题最简单的方法就是读缺失等待，直到写缓存为空为止。另外一个方案是在读缺失时查看写缓存的内容，如果没有冲突且存储器系统可以访问，就让读缺失继续。实际上，所有的桌面处理器和服务器都采用了后一种方法，使读缺失优先于写缺失。

在写回法的 Cache 中，处理器的写开销也可以降低。假定发生读缺失，一个存储器脏块将被替换。我们可以把脏块复制到一个缓存区后就去看下一层存储器，然后再把该脏块写入下一层存储器，而不是把脏块先写入到下一层存储器中，然后再对下一层存储器进行读操作。这种方法使得处理器读操作很快结束（通常处理器很可能因为读操作而等待）。与上面的情况类似，如果读缺失发生，处理器可以停顿直至缓存区为空，或是检查缓存区中字的地址来看看是否发生冲突。

以上已经介绍了五种降低缺失率或缺失代价的优化技术，现在介绍降低平均存储器访问时间公式组成中的最后一项。命中时间很关键，因为它会影响处理器的时钟频率；目前的许多处理器的 Cache 访问时间受时钟频率所限，即使处理器需要多个时钟周期来访问 Cache。因此，快速的命中时间对于平均存储器访问时间公式是非常重要的。

第六种优化技术：在 Cache 索引过程中避免地址转换来减少命中时间

即使容量小、结构简单的 Cache 也避免不了从处理器的虚拟地址到存储器物理地址的变换。正如下面的 C.4 节将要讲到的，处理器将内存视为存储器层次结构的另外一级，硬盘上的虚拟存储器的地址必须映射到内存中。

因为命中相对于缺失来说是经常性事件，因此为了“加快经常性事件的速度”，在 Cache 中需要使用虚拟地址。这样的 Cache 称为**虚拟 Cache**，而使用物理地址的传统 Cache 称为**物理 Cache**。区分 Cache 检索和地址比较是十分重要的，因此，问题是用虚拟地址还是物理地址来检索 Cache，标志字段是用虚拟地址还是物理地址做比较。完全采用虚拟地址免除了 Cache 命中时的地址变换时间。那么，为何设计者不都选用虚地址 Cache 呢？

第一个原因是保护问题。在虚拟地址变换为物理地址时，必须进行页级保护检查。一种解决办法是在发生缺失时，将 TLB 的保护信息复制下来，给它增加一个字段，每次访问虚拟地址 Cache 时只要检查该字段即可。

另一个原因是每次进程切换时，虚拟地址对应着不同的物理地址，这要求刷新 Cache。图 C.16 给出了这种刷新对缺失率造成的影响。一种解决办法是增加 Cache 地址标识符字段的宽度，给它加上一个进程标识符（PID）。如果操作系统把这些标识符分配给了进程，它仅需要在回收 PID 时刷新

Cache；也就是说，PID用来确定Cache中的数据是否属于该程序。图C.16说明了通过使用PID避免Cache刷新对缺失率的改善。

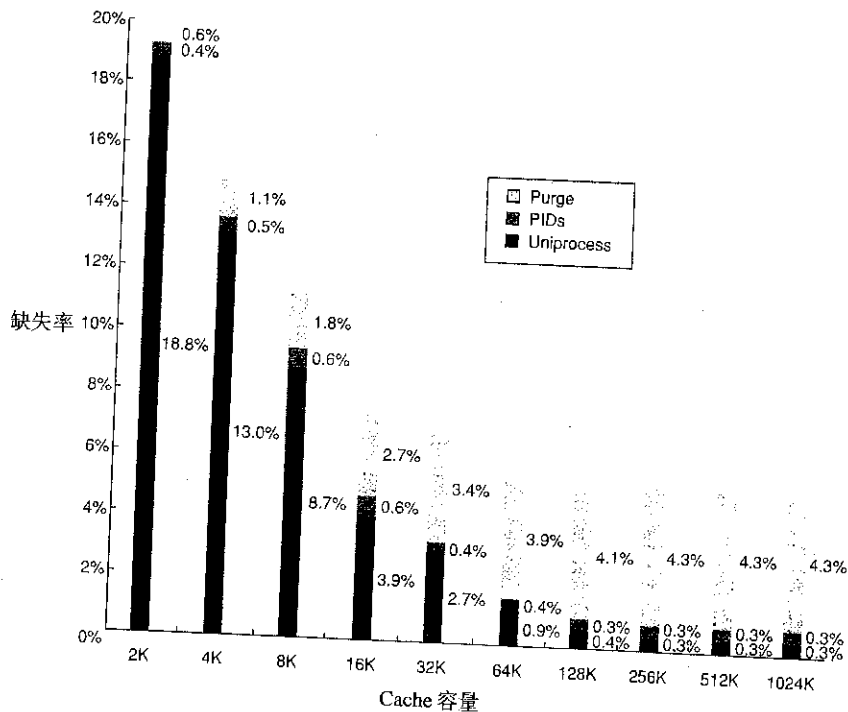


图 C.16 在三种不同情况下，缺失率和虚拟地址 Cache 容量的关系：不切换进程（单进程）、利用 PID 进行进程切换和无 PID 时进程切换（称为 purge 方式）。采用了 PID 时，单进程的缺失率增加 0.3%~0.6%；而 purge 方式使缺失率降低了 0.6%~4.3%。Agarwal[1987]在 VAX 上运行 Utrix 操作系统收集到这些数据，其中 Cache 采用直接映射方式，块容量为 16 字节。注意，Cache 容量从 128 KB 增加到 256 KB，缺失率增大：这是由于当改变 Cache 容量时，也改变了存储器块与 Cache 块的映射方式，从而改变了冲突缺失率。

第三个导致虚拟 Cache 未能更流行的原因是，对于同一物理地址，操作系统和用户程序可以使用两个不同的虚拟地址。这些双重地址称为同义或别名地址，会导致同一数据在虚拟 Cache 中存在两个副本：如果其中的一个被修改了，另一个的值将会是错误的。而物理 Cache 则不存在这类问题，因为上述访问将首先变换到相同的物理 Cache 块中去。

一种称为别名消去的硬件技术可以保证每一个 Cache 块有一个唯一的物理地址。Opteron 的指令 Cache 容量为 64 KB，一页为 4 KB，2 路组相联，硬件必须处理组索引字段中三个虚拟地址位所涉及的别名。具体做法是：发生缺失时检查 8 个所有可能的位置——四组，每组两块——保证没有与已经取得的数据的物理地址冲突。如果冲突，使之无效，故保证了新取到 Cache 的数据的物理地址是唯一的。

使别名共享某些地址位的软件方法可以使这个问题的解决变得容易些。例如，Sun 公司的 UNIX 版本所有别名地址的后 18 位相同：这种限制称为页着色（page-coloring）。注意，页着色适用于虚拟存储器采用简单组相联映射的情形：4 KB (2^{12}) 页采用 64 (2^6) 路组相联映射保证了物理地址与虚拟地址的后 18 位相匹配，这种限制意味着采用直接映射的 Cache，若容量小于等于 256 KB (2^{18})，则不可能产生双重的物理地址。从 Cache 的角度看，软件保证虚拟地址和物理地址最后几位相同的同时，页着色显著地增加了页内偏移。

最后一个涉及到虚拟地址的是I/O。I/O主要使用物理地址，因此，需要将物理地址映射为虚拟地址以便与虚拟Cache进行交互作用（I/O对Cache的影响将在第6章中进一步讨论）。

充分利用虚拟Cache和物理Cache的一种方法是使用页偏移——虚拟地址和物理地址中相同的部分——去索引Cache。当利用这个索引进行读Cache时，地址中的虚拟部分被转换，标志字段匹配使用物理地址。

这种方法允许Cache读立刻开始，并且标志比较使用物理地址。这种“虚拟索引、物理标记”的方法有一个限制：采用直接映射的Cache，其容量不能大于一页的大小。比如，图C.5中的数据Cache，索引字段为9位，块内偏移为6位。若要使用这种方法，虚拟页大小至少是32 KB或 2^{9+6} 字节。否则索引字段的一部分将由虚拟地址转化为物理地址。

采用多路组相联既可以保证索引位较少从而能够从地址的物理部分获得，又可以使Cache的容量较大。如前所述，索引大小由下面的公式确定：

$$2^{\text{Index}} = \frac{\text{Cache大小}}{\text{块大小} \times \text{组相联度}}$$

例如，将相联度和Cache容量同时增加一倍，就不会改变索引字段的位数。Pentium III的页大小为8 KB，Cache容量为16 KB，使用2路组相联，避免了转换。作为一个极端的例子，IBM 3033 Cache采用16路组相联。尽管研究表明，采用8路以上组相联的缺失率并没有比8路组相联时的缺失率有多少改善。采用相联度较高的16路组相联映射，可以使64 KB大小的Cache使用物理索引编址，而不受IBM系统结构中4 KB页大小的限制。

基本Cache优化总结

这一节中改善命中时间、带宽、缺失代价及缺失率的多种技术通常会对平均存储器访问时间公式中的其他因子产生影响，同时也会影响存储器层次结构的复杂性。图C.17总结了这些技术，并对这些技术对存储器层次结构复杂性的影响进行了评价，使用“+”表示该技术改善相应特性，使用“-”表示该技术会对相应特性有不利影响，空白表示没有影响。我们也注意到，几乎没有任何一种技术能够改善一种以上的性能指标。

技术	命中时间	缺失代价	缺失率	硬件复杂度	备注
增大块容量		-	+	0	效果较小，Pentium 4 二级Cache使用128字节
增大Cache容量	-		+	1	广泛使用，特别是在二级Cache中
增加相联度	-		+	1	广泛使用
多级Cache		+		2	昂贵的硬件；如果一级Cache块大小≠二级Cache块大小，则较难；广泛使用
读缺失优先于写		+		1	广泛使用
避免索引Cache 时的地址转化	+			1	广泛使用

图C.17 基本Cache优化技术总结以及各种技术对Cache复杂度和性能的影响。通常一种技术只改善某一个因子。“+”表示该技术改善相应特性，“-”表示该技术会对相应特性有不利影响，空白表示没有影响。存储层次结构复杂度是主观定义的，0表示最简单，3表示最复杂

C.4 虚拟存储器

……系统被设计成将主存储器与后备存储器组合在一起,在程序员看来好像只有一级存储,必须进行的地址变换是自动完成的。

Kilburn 等[1962]

计算机运行的任何时刻都存在多个进程,每个进程都有自己的地址空间(进程在下一节讲述)。如果为每个进程分配全部的地址空间,那么系统开销太大,而且对于很多进程来说,它们只不过是使用了地址空间的一小部分。因此,可以把物理存储器的一部分拿出来让许多进程共享。

虚拟存储器就是这样一种存储器共享技术,它把物理存储器分块并分配给不同进程使用。显然,在这种技术中必须有一套**保护机制**,即只能让一个进程访问属于它自己的内存块。虚拟存储器技术通常也能减少程序启动时间,因为程序运行之前不需要把所有代码和数据都载入物理存储器。

虚拟存储器的内存保护功能对于当代计算机来说是至关重要的,但是存储器共享并不是虚拟存储器的设计目标。如果一个程序的自身大小比物理存储器还大,那么程序员就需要想些办法加以解决。以前的解决方法是把程序分成很多片段,同时标明哪些片段是独占的,这样就可以在程序执行过程中加载需要的片段,卸载不需要的片段。可想而知,这项额外的负担大大降低了程序员的效率。

这种情况下人们发明了虚拟存储器,它将程序员从这种负担中解脱出来:它自动管理一个由内存和二级存储器(也称辅存)组成的两级存储系统。图 C.18 所示为在虚拟存储器到物理存储器的一个拥有四个页面程序的地址映射图。

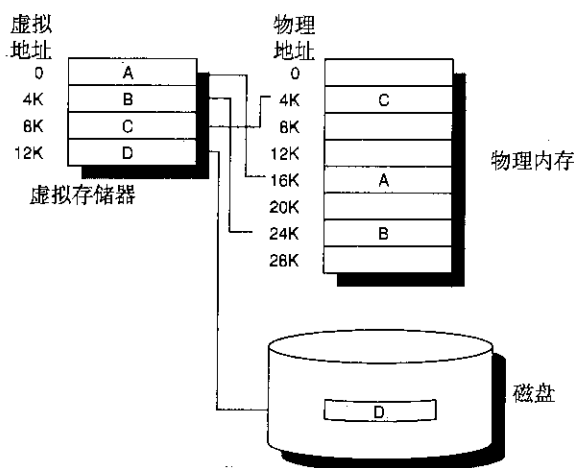


图 C.18 左边是拥有连续虚拟地址空间的程序。它包含四个内存页: A, B, C, D。其中三页在物理内存中,剩下一页在磁盘上

除了共享存储器和自动管理存储器层次结构,虚拟存储器还使程序的加载变得更加简单。**重定位**技术可使一个程序在物理存储器的任何位置运行。图 C.18 中的程序可以被放置在物理存储器或是磁盘的任何地方,只需要改变映射关系即可(在虚拟存储器流行以前,处理器中有一个重定位寄存器,专门用于这种操作)。我们也可以用软件实现这样的操作,它使得一个程序在每次运行时都改变它的地址。

第 1 章中提到的几个一般性的 Cache 层次概念和虚拟存储器其实是相似的,虽然它们的术语不太一样:不管是页还是段,都是一种分块方式;不管是页失效还是地址失效都是指缺失。在虚拟存

存储器系统中,处理器产生虚拟地址,然后通过硬件、软件进行一系列的转换,就可以得到可以实际访问的物理地址了。这个过程称为存储器映射或者地址变换。目前的虚拟存储系统中,两级存储器是由动态随机存储器 DRAM 和磁盘存储器组成的。图 C.19 列出了一些虚拟存储器的参数指标。

除了图 C.19 所示的定量值,Cache 和虚拟存储器之间还有以下不同:

- Cache 缺失时主要由硬件控制进行替换操作,而虚拟存储器的替换主要由操作系统控制。因为一次缺失会造成很长的延迟,所以选择一个好的替换算法是很重要的,操作系统能够担负这个责任,它花费一些时间决定替换出哪些信息。
- 处理器的地址大小决定了虚拟存储器的大小,但和 Cache 的大小无关。
- 辅存除了作为存储器层次中内存的下一级后备存储器外,还被用于文件系统,文件系统通常不是地址空间的一部分,但占据了辅存的大部分空间。

参数	一级 Cache	虚拟存储器
块(页)大小	16~128 字节	4096~65 536 字节
命中时间	1~3 个时钟周期	100~200 个时钟周期
缺失代价	8~200 个时钟周期	1 000 000~10 000 000 个时钟周期
访问时间	(6~160 个时钟周期)	(800 000~8 000 000 个时钟周期)
传输时间	(2~40 个时钟周期)	(200 000~2 000 000 个时钟周期)
缺失率	0.1%~10%	0.000 01%~0.001%
地址映射	25~45 位物理地址到 10~20 位 Cache 地址	32~64 位虚拟地址到 25~45 位物理地址

图 C.19 Cache 和虚拟存储器的典型参数指标。虚拟存储器的相应参数是 Cache 的 10 倍到 100 000 倍。一级 Cache 容量至多是 1 MB,物理存储器是 256 MB 到 1 TB

虚拟存储器还包括其他一些相关的技术。虚拟存储系统可以分为两类:一类是使用固定大小的分块,称为页;另一类是使用可变长度的分块,称为段。页大小固定在 4096 到 8192 字节之间,而段大小可以变化。处理器支持的最大段范围从 2^{16} 字节到 2^{32} 字节不等,最小的段可以只有 1 字节。图 C.20 显示了这两种方法是如何划分代码和数据的。

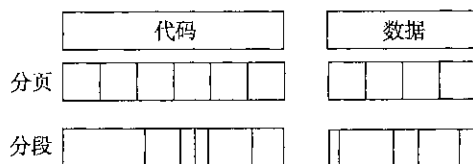


图 C.20 分页和分段情形下程序划分的例子

决定是使用页式虚拟存储器还是使用段式虚拟存储器会对处理器产生影响。页式编址只有一种定长地址,地址位数固定,分成页号和页内偏移两部分,与 Cache 编址相似。对于段式编址,单一地址类型是不够的:由于段的大小会变化,要求一个字表示段号,另一个字表示段内偏移,共需要两个字。非段式的地址空间对于编译器来说是比较简单的。

这两种方法的优缺点在操作系统的教材中都有详细的介绍,图 C.21 给出了一个总结。由于存在替换问题(图中第 3 行),现代计算机中很少使用纯粹的分段技术。有些计算机将两者结合起来,这称为段页式,就是说一个段是由若干个页组成的。由于不必连续存储,从而简化了替换操作,并且也不需要整个段都存放在内存中。最近采用的一种混合方法是提供多种容量的页,较大的页面大小是最小页面的大小乘以 2 的幂。比如 IBM 405CR 嵌入式处理器所使用的页面大小有 1 KB, 4 KB($2^2 \times 1$ KB), 16 KB($2^4 \times 1$ KB), 64 KB($2^6 \times 1$ KB), 256 KB($2^8 \times 1$ KB), 1024 KB($2^{10} \times 1$ KB) 和 4096 KB($2^{12} \times 1$ KB)。

	页式	段式
地址中的字数	一个	两个（段地址和偏移地址）
是否对程序员可见	不可见	有可能可见
替换一个块的开销	小（所有的块可以相同大小）	大（必须在内存里寻找连续的、大小不同和未使用的部分）
未有效使用的存储器	内部碎片（页内未使用部分）	外部碎片（内存中未使用部分）
有效磁盘通信	是（可以调整页大小，从而平衡访问时间和传输时间）	不一定（一个很小的段可能只要传送几个字节）

图 C.21 分页和分段比较。两种方法都会浪费存储器，浪费的多少取决于块的大小，以及段大小和内存大小的关系。非限制指针类型的编程语言既需要指定段地址，也需要指定偏移地址。段页式方法是两者的结合，所以兼具两者的优点：段由页组成，因此块的交换变得简单，而且一个段也可被视为一个逻辑单位

关于存储器层次结构的四个问题

我们下面准备回答关于虚拟存储器层次的四个问题。

Q1：一个块可以放在内存中的什么位置？

由于虚拟存储器缺失后需要读取磁盘存储设备，因此缺失后的延迟非常高。假如只能在较低的缺失率和简单的替换算法两者之间选择其一，则操作系统的设计者通常会避免缺失代价过高而选择较低的缺失率。因此，操作系统允许将块放在内存中的任意位置。引用图 C.2 的术语，这种策略称为全相联。

Q2：如何在内存中查找一个块？

页式和段式都依赖于一个按页号或段号索引的数据结构。这个数据结构包含块的物理地址。对于段式虚拟存储器，偏移与段的物理地址相加得到最终的物理地址。对于页式虚拟存储器，偏移与物理页地址只需简单地拼接即可得到最终的物理地址（见图 C.22）。

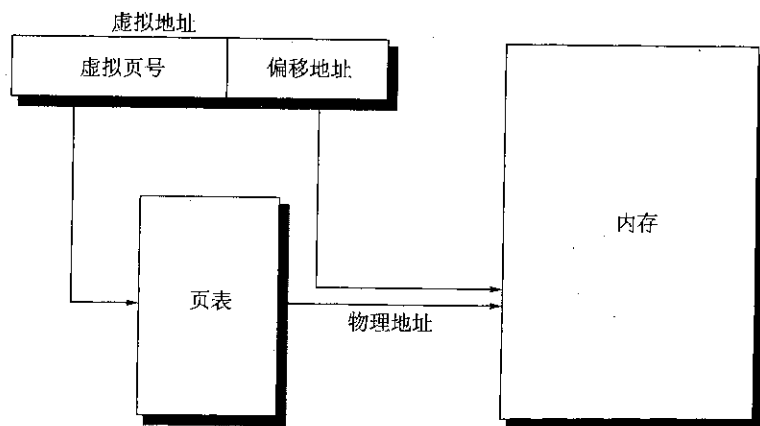


图 C.22 虚拟地址通过页表向物理地址的映射

包含物理页地址的数据结构通常采用页表的形式，它按照虚拟页号索引，页表的大小为虚拟地址空间中的页数。如果虚拟地址为 32 位，页大小是 4 KB，页表的每项是 4 个字节，那么页表的大小就是 $(2^{32}/2^{12}) \times 2^2 = 2^{22}$ 字节，即 4 MB。

为使这个数据结构尽可能小，一些机器中应用了一个虚拟地址的散列函数使得这个数据结构的大小仅为内存中实际物理页的数量；这比虚拟页数小得多，这样的数据结构称为反向页表。在上面

的例子中,一个 512 MB 的物理存储器只需要 1 MB ($8 \times 512 \text{ MB}/4 \text{ KB}$) 空间来存放反向页表;多出的 4 字节用来存放虚拟地址。HP/Intel 的 IA-64 处理器同时支持页表和反向页表,由操作系统设计者来决定使用哪个页表。

为减少地址变换时间,计算机使用了一个 Cache 用于该地址转换,称为变换旁视缓冲器 (TLB),或简单称为变换缓存,将在后面详细讲述。

Q3: 当发生虚拟存储器缺失时应该替换哪一个块?

在前面已经提到,操作系统的基本原则是尽可能减少页缺失。按照这一原则,几乎所有的操作系统都试图替换出最近最少使用的块 (LRU),因为根据历史记录,它是将来最不可能再被访问的块。

为帮助操作系统确定最近最少使用的块,许多机器都提供了一个使用位或者参考位,它是一个逻辑集合,当某一页被访问时置位 (为减少工作量,只有当转换缓冲缺失时才置位)。操作系统周期性地清除使用位随后记录它们,因此可以确定在特定的时间周期内涉及到了哪些页。通过这种方式保持跟踪,操作系统可以选择出最近最少使用的一页。

Q4: 写操作时会发生什么?

内存的下一级存储器是旋转式磁盘存储器,对它的访问要花费数百万个时钟周期。由于内存与磁盘的访问时间相差太大,没有一个虚拟存储器操作系统采用写直达策略,即处理器每一次将数据写入内存的同时也写入磁盘。因此,写策略总是采用写回法。

由于对下级磁盘存储器冗余访问的代价过高,虚拟存储器系统通常包含一个重写位来标识出那些应该被写回磁盘的块,这些块即是那些从磁盘装入存储器后被修改过的块。

快速地址转换技术

页表通常太大了,以至于需要存放在内存中,有时对页表本身还要分页。这意味着要从存储器中访问一个数据 (取指令、读数据或写结果) 至少要访问存储器两次,一次访存获得物理地址,另一次访存获得数据。如第 5 章提到的,我们利用了局部性原理来避免冗余的存储器访问。通过将一些地址转换保存在一个专门的 Cache 中,可以使得一个存储器访问很少要求第二次访存。这个特殊的地址变换 Cache 称为变换旁视缓冲器 (TLB),也称为快表 (TB)。

TLB 中的项与 Cache 项相似,标志字段包含部分虚拟地址,数据部分包含物理页号、保护字段、有效位,通常还有一个使用位及一个重写位。如果要改变页表中某一项的物理实页号或保护页表中某一项的状态,则操作系统必须确认原来的项不在 TLB 中;否则,系统将不能正确工作。注意重写位表示相应的页是脏的 (被修改过),而非 TLB 地址转换是脏的,也不表示数据 Cache 中的特定块是脏的。操作系统如果要改变这些位,它会首先改变页表中的值,然后通知 TLB 中相应的项无效。当这一项重新从页中装载时,TLB 中的内容就和页表内容一样了。

图 C.23 给出了 Opteron 数据 TLB 的结构,每一个转换步骤由数字标出。TLB 采用全相联映射,转换过程从将虚拟地址发送至所有的标志开始 (步骤 1 和步骤 2)。当然,进行比较的标志必须是有效的。同时,要检查存储器访问类型是否与 TLB 中的保护信息冲突 (也在步骤 2 进行)。

考虑到与 Cache 类似的原因,TLB 中不必包含 12 位页偏移。匹配标志通过 40:1 多路选择器发出相应的物理地址 (步骤 3)。然后,页偏移与物理页号拼接组成完整的物理地址 (步骤 4)。物理地址为 40 位。

地址转换是决定处理器时钟周期的重要因素,所以 Opteron 使用了一个虚拟地址和物理标注的一级 Cache。

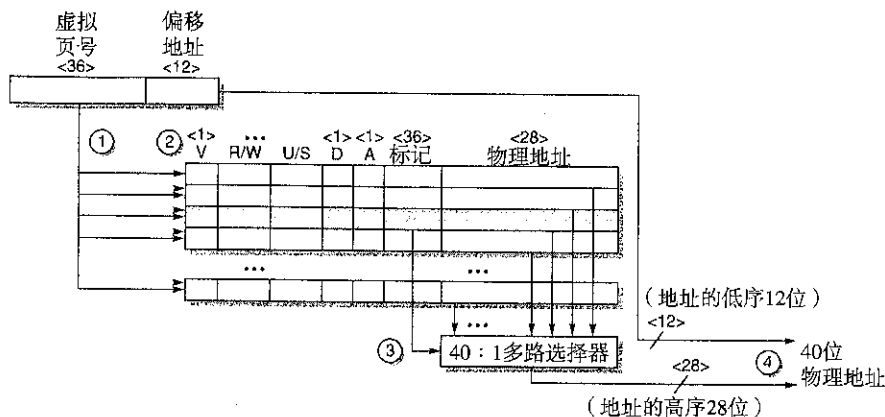


图 C.23 Opteron 中数据 TLB 的地址转换操作过程。此过程的四步用圆圈中的数字标示。TLB 有 40 个项。C.5 节描述了 Opteron 页表项的保护和访问机制

选择页大小

最直观的系统结构参数是页大小。选择页大小就是要在选择较大的页还是较小的页之间进行权衡。下面是选择较大页的一些理由：

- 页表的大小与页大小成反比，选择较大的页能够节省页表存储器（或其他用于存储器映射的资源）。
- 在 C.3 节中提到，较大页允许大 Cache 有较短的命中时间。
- 在存储器和辅存间（有时可能通过网络）传送一个较大页比传送较小页的效率要高。
- TLB 项个数受限制，所以较大页意味着更多的存储器信息被有效映射，因此可以减少 TLB 缺失次数。

正是由于最后一个原因，近来的许多微处理器均支持多种页大小；对许多程序来说，TLB 缺失对 CPI 的影响与 Cache 缺失对 CPI 的影响可能同样重要。

选择较小页的主要目的是节省存储空间。当虚拟存储器中一个连续区域的容量不等于页大小的倍数时，若页容量较小，则被浪费的存储空间较少。页内未被使用的存储空间称为内部存储碎片。假设每个进程有三个基本的段（文本段、堆段和栈段），则每个进程平均浪费存储空间为页大小的 1.5 倍。对于存储空间为成百上千兆字节且页大小在 4 KB 到 8 KB 之间的机器来说，这种浪费数量是可以忽略的。当然，当页容量变得非常大时（大于 32 KB），很多存储空间（内存和辅存）可能被浪费，当然也浪费了 I/O 带宽。最后关注的一个问题是进程启动时间；许多进程较小，所以较大的页将延长调用一个进程的时间。

虚拟存储器和 Cache 的小结

因为虚拟存储器、TLB、一级 Cache 和二级 Cache 都要进行虚拟地址和物理地址的转换，因此可能会造成一些理解上的混乱。图 C.24 是通过两级缓存系统将 64 位虚拟地址转化为 41 位物理地址的例子。一级 Cache 采用虚拟索引，物理标记。因为 Cache 大小和页大小都是 8 KB。二级 Cache 大小为 4 KB。两个 Cache 的块大小都是 64 字节。

首先，64 位的虚拟地址逻辑上被分为两部分：虚拟页号和页内偏移。前者被送到 TLB 中转换为物理地址，后者的高位被发送到一级 Cache 中用做索引。如果 TLB 命中，那么物理页号会送到

一级 Cache 进行比较。如果比较相等,那么一级 Cache 命中。之后,将根据块偏移选出需要的字发送给处理器。

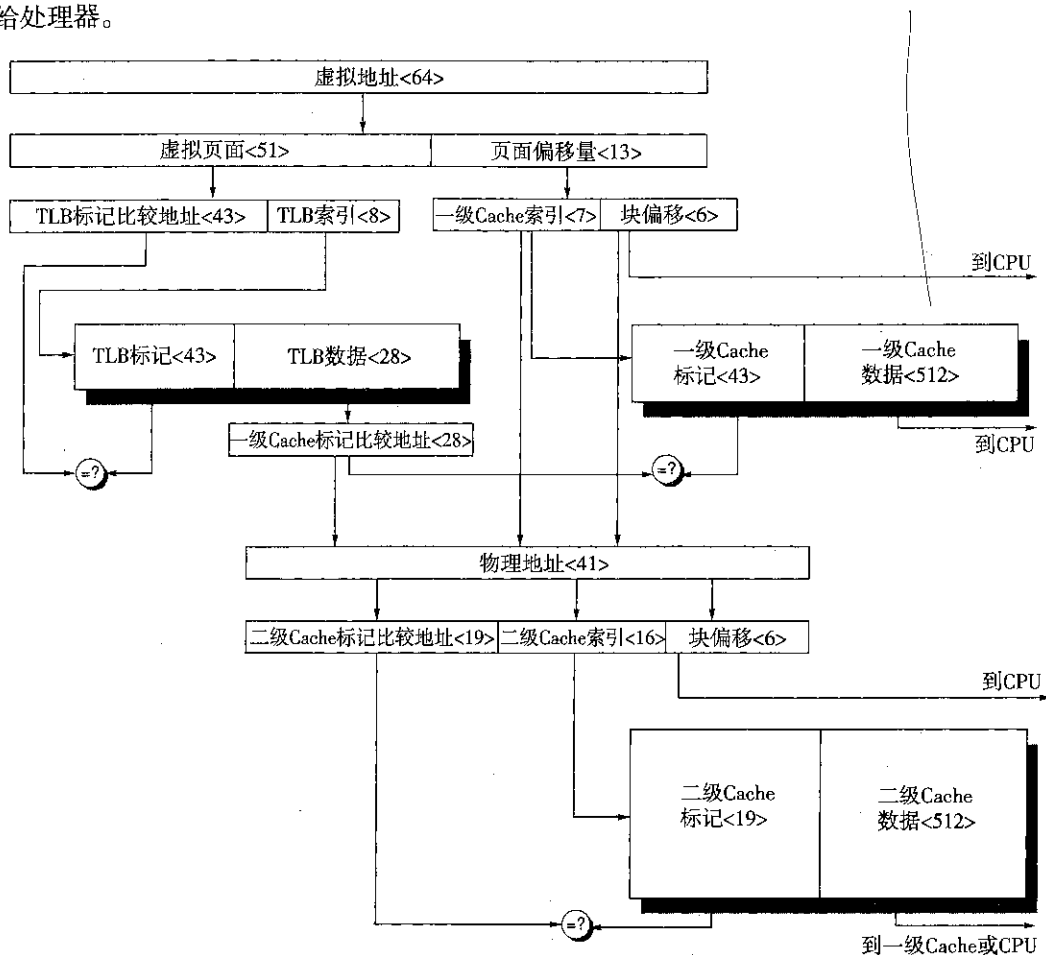


图 C.24 一个假想的存储器层次结构中虚拟地址到二级 Cache 的访问过程。页大小是 8 KB。TLB 使用直接映射,有 256 个项。一级 Cache 是一个 8 KB 大小的直接映射,二级 Cache 是一个 4 MB 的直接映射。两个 Cache 的块大小都是 64 字节。虚拟地址为 64 位,物理地址为 41 位。该简单图示与实际 Cache 的主要区别,是实际 Cache 中有多个图示中的组成部分

如果一级 Cache 缺失,物理地址将被继续用于查询二级 Cache。物理地址的中间部分被用做 4 MB 二级 Cache 的索引。然后相应的二级 Cache 的标记将和物理地址的高位部分比较。如果相等,那么二级 Cache 命中,将根据块偏移选出需要的数据送往处理器。如果二级 Cache 缺失,物理地址将被用来访问存储器中的块。

图 C.24 只是一幅简化的图,实际的 Cache 中有多个相同单元。这幅图中只有一个一级 Cache,而实际上有两个,也就是上图的上半部需要重复一次。所以,通常也会有两个 TLB: 一个是受寄存器 PC 驱动的指令 TLB,另一个是受当前有效地址驱动的数据 TLB。

图中的第二个简化是所有的 Cache 和 TLB 都是直接映射的。假如其中任何一个采用 n 路组相联,我们就需要把每一个标记存储器、比较器和数据存储器复制 n 次,并且用一个 $n:1$ 多路复用器连接数据存储器以取得一个命中结果。当然,如果 Cache 的大小总和不变,根据图 C.7 的公式,它的索引可以减少 $\log_2 n$ 位。

C.5 虚拟存储器的保护和实例

多道程序设计的出现,使得计算机被并行运行着的多个程序所共享。在这种情形下,提出了新的需求,即在程序之间需要提供保护和共享机制。这与现在计算机的虚拟存储器技术紧密相关,下面将通过两个实例讲述这一主题。

多道程序设计产生了**进程**的概念,进程可以比喻为一个程序呼吸的空气和存活的空间——即一个运行着的程序连同持续运行它所必须的所有状态。分时系统是多道程序的变体:多个交互式用户同时共享着处理器和存储器,所有用户都感觉到好像是拥有自己的计算机。因此,在任何时候,必须能够从一个进程切换到另一个进程,这称为**进程切换**或**上下文切换**。

无论进程是从开始到结束连续地执行,还是在中间反复地被打断并切换到其他进程,都必须被正确执行。确保进程正确运行的责任由计算机的设计者和操作系统设计者共同承担:计算机设计者必须确保进程中有关处理器的状态信息被保存,且可以恢复;操作系统的设计者须确保进程不被其他的计算所干扰。

保护一个进程状态的最安全的方法是将当前信息复制到磁盘,但这样进程切换将花费几秒钟——这对一个分时环境来说时间太长了。

这个问题可通过以下方法解决:操作系统将内存划分,使同一时刻不同的进程在存储器中都有自己的状态信息。这意味着操作系统的设计者需要计算机设计者的帮助,提供进程保护使得一个进程不能修改其他的进程。除此之外,计算机同时也为提供多个进程提供代码和数据共享,允许进程间通信或通过减少同一信息的冗余副本而节省存储器。

保护进程

每个进程都有属于自己的页表,指向存储器中不同的页,以此实现相互间的保护。显然,用户进程不能修改自己的页表,否则进程间的保护机制将被破坏。

如果计算机的设计者和用户对进程保护有所顾虑,则保护可以被加强。在处理器保护结构中加入**环状保护**可将两级(用户和内核)存储器访问保护扩展到更多级。就像军方具有绝密、机密、秘密、非秘密环状安全分类的系统,安全级中最中心的环允许最受信任的进程访问任何信息,其次受信任的进程可访问内核级以外的任何信息,依此类推,直到普通程序,由于它是最不受信任的进程,所以访问范围最受限制。也可以对存储器中哪一块可以包含代码——执行保护——甚至对于两级之间的入口点进行限制。采用了环状保护的Intel 80x86保护结构将在本节的后面讲述。现在看来,环状保护在实际应用中是否能够比简单的用户和内核模式保护系统有所改进还不清楚。

如果设计者对安全问题有更高的要求,则最简单的环可能是不够的。系统需要一个新的分类模式以限制密级最高程序的自由度。它不像军方模式,而是钥匙和锁的关系:一个程序除非有钥匙,否则就不能打开数据的访问锁。为使这些钥匙(或称为**权限**)有效,硬件和操作系统必须能够明确地将它们从一个程序传到另一个程序,而不允许程序自身生成它们。如果要使检查钥匙的时间尽可能短,则必须有大量硬件的支持。

80x86系统结构近年来尝试了许多类似的方法。由于这种系统结构支持向后兼容,因此这种系统结构的新版本就包含了它在虚拟存储器上的最新探索。我们将在此复习两种方法:一是较老的段式地址空间,二是平面、更新的64位地址空间。

段式虚拟存储器示例：Intel Pentium 保护模式

对设计者来说，第二代系统是最危险的……很容易趋向于把第一代系统中谨慎使用的一些思想，在第二代系统的设计中都套用上。

F. P. Brooks, Jr.

The Mythical Man-Month(1975)

先前的8086采用段式编址，但它并没有提供虚拟存储和保护。段有基址寄存器但没有界址寄存器和访问检查，在段寄存器被读取之前，相应的段必须在物理存储器中。Intel致力于虚拟存储和保护，这在8086后续产品（今天我们称之为IA-32）中得到了体现，通过扩展一些字段以支持较大的地址范围。这种保护模式设计得非常精巧，它认真考虑了许多细节问题以避免安全漏洞。下文将着重讲述Intel的一些保护措施。如果这部分内容读起来难以理解，你就可以想象实现它们是多么困难！

首先是对传统的二级保护模式加倍增强措施，Pentium有四级保护：最内一级(0)对应于传统的内核模式，最外一级(3)对应于传统的用户模式。IA-32每一级都有独立的堆以避免两级之间的安全破坏。它也有类似于传统的页表的数据结构，包含段的物理地址和针对转换地址的检查列表。

Intel的设计者在这方面的探索一直没有停止。IA-32划分地址空间，允许操作系统和用户访问全部空间。在此空间内，IA-32用户能调用一个操作系统进程，甚至在保留全部保护状态时向系统进程传递参数。由于操作系统堆栈和用户堆栈不同，安全的调用较复杂。IA-32甚至允许操作系统为传递给它的参数保持被调用进程的保护级别。IA-32不允许用户进程请求操作系统进而间接地访问一些它自己不能访问的空间，这弥补了潜在的保护漏洞（这种安全漏洞称为特洛伊木马）。

Intel的设计者本着尽量少地信任操作系统的原则支持共享和保护。下面是一个应用这种受保护共享的例子。假设一个工资单程序要写出工资单、更改关于工资支付的时间信息，这样这个程序应该能够读出工资标准和每年每天的支付信息，也能更改每年每天的支付信息，但不能更改工资标准。稍后，我们将看到支持这一特性的机制。在本节余下部分，我们将对IA-32保护模式进行形象的描述，以便对这种改进加深理解。

加入边界检查和存储映射

增强处理器性能的第一步是使段式编址既提供基址也提供边界检查。与8086中的基址不同，IA-32中的段寄存器包含一个索引，指向一个称为描述符表的虚拟存储数据结构。描述符表与传统页表的作用相同。在IA-32中一个段描述符等同于页表的一个项，它包含PTE中的一些字段：

- **当前位**：等同于PTE中的有效位，用来指示这是一个有效转换。
- **基址域**：等同于页号地址，包含段中第一个字节的物理地址。
- **访问位**：类似于一些系统结构中的访问位或使用位，用于替换算法。
- **属性域**：为使用该段的操作指明有效操作和保护级。

还包括一个范围字段，这在页式系统中是没有的，它确定该段有效偏移的上界。图C.25为IA-32段描述符的示例。

除了段式编址外，IA-32提供了一个可选的页式系统，其中32位地址的高位部分选择段描述符，中间部分用做索引指向根据描述符选择的页表。下面我们讨论不依赖于页的保护系统。

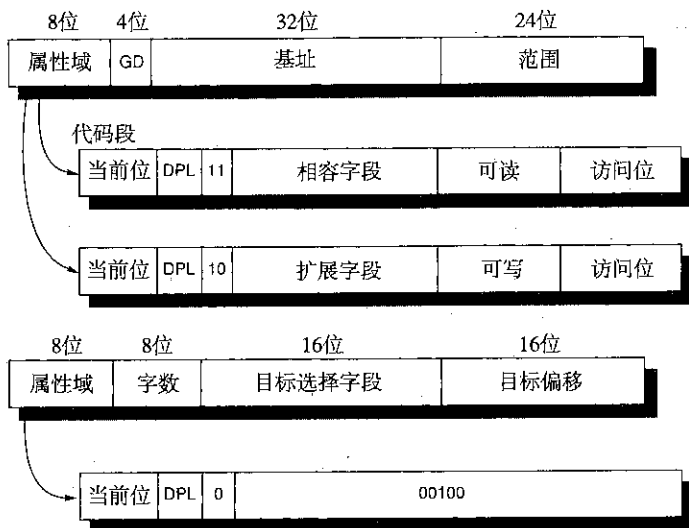


图 C.25 IA-32 段描述符由属性字段中的位来区分。基址、范围、当前、可读、可写均无须解释。D 给出指令的默认编址范围：16 位或 32 位。G 给出段范围的粒度：0 表示在字节级，1 表示在 4 KB 页一级。当启动页面调度设置页表大小时，G 被置为 1。DPL 表示段描述符特权级别：用于与代码特权级别进行对照检查，以确定该访问是否被允许。相容字段表明代码具有被调用代码的特权而不具有调用者的特权，它用于库例程。向下扩展字段反转控制 (flips the check)：基址字段表示高地址，范围字段表示低地址，它适用于向下增长的栈段。字数字段控制调用通道中从当前栈复制到新栈中的字数。调用通道段描述符的另两个字段，即目标选择字段和目标偏移，分别选择要调用目标段的描述符和内部偏移。在 IA-32 中，除了这三个段描述符外还有更多的段描述符

加入共享和保护

为了提供受保护的共享，一半地址空间被所有进程共享，而另一半对每一个进程都是独占的，它们分别称为全局地址空间和局部地址空间。每一半都被赋予一个带有适当名字的描述符表：指向共享段的描述符放在全局描述符表中；而私有段的描述符放在局部描述符表中。

程序利用指向描述符表的索引和标识所需表的位，来读取 IA-32 段寄存器内容。依照描述符的属性字段用来对操作进行检查，如果偏移比范围字段小，则将处理器中的偏移地址与描述符中的基址相加得到物理地址。每个段描述符都有单独的两为字段描述这个段的合法访问级别，只要程序试图使用一个段描述符中更低保护级别的段，就发生错误。

现在分析怎样调用上面提到的工资单程序以便更改每年每天的信息，但不允许更改工资。程序可获得一个数据信息的段描述符，其中的可写字段被清零，这意味着程序仅能读数据，但不能写数据。然后提供一个可信程序，只能对每年每天的信息进行写操作，该程序获得一个可写字段被置 1 的段描述符（见图 C.25）。工资程序使用一个代码段描述符调用这段带有相容字段的可信代码。这意味着被调程序具有被调用代码的特权级别，而不具有调用程序的特权级别。因此，工资单位程序能读出工资，并通过调用可信程序更改每年每天的信息，但不能更改工资。如果一个特洛伊木马在该系统中存在，它只能位于可信任代码中才能生效，其唯一能做的工作是更改每年每天的信息。这种类型保护采用的是限制受攻击的范围的方法以增强安全性。

增加从用户到操作系统通路的安全调用和继承参数的保护级别

允许用户进入操作系统是大胆的一步。那么，硬件设计者怎样才能在不依赖操作系统和其他代码的前提下增加系统的安全系数呢？IA-32采用的方式是对用户能进入代码的范围进行限制，将用户参数安全地放置在合适的栈中，确保用户参数不能获得被调用代码的安全级别。

为了限制进入其他的代码，IA-32提供了一个特别的段描述符，称为调用通路，由属性字段中的一位来标识。与其他的段描述符不同，调用通路是存储器中一个对象的全物理地址；处理器提供的一位来标识。这正如前面所述，其目的在于阻止用户通过随机跳转进入一个受保护的或具有更高私密性要求的代码段，在本例中，这意味着工资单程序能调用可信代码的唯一地方是合适的边界。要使相容段正常工作则需要这种限制。

如果调用者和被调用者“相互怀疑”，以至于没有一方信任其他一方，将会发生什么呢？解决的方法可以在图 C.25 下部段描述符的字数字段中发现。当一个调用指令请求一个调用通路描述符时，描述符将标识符中的字数从局部栈复制到符合这个段级别的栈中。这就允许用户通过首先将参数压入局部堆栈来传递参数，然后硬件安全地将它们传递到正确的堆栈。从调用通路返回时，将两个堆栈中的参数都弹出并将任何返回值复制到合适的堆栈。注意这种模式与实际应用中采用寄存器传递参数的方法是不兼容的。

这种方式仍然存在潜在漏洞，如果操作系统使用作为参数传递的用户地址，它将具有操作系统的安全级别而不具有用户的安全级别。IA-32通过在每一个处理器段寄存器中使用两位表示被请求的保护级别来解决这个问题。当一个操作系统进程被调用时，它可以执行一条指令，将所有地址参数中的这两个位设置成调用该系统进程的那个用户进程的保护级别。这样，当这些地址参数装入寄存器中时，它们将把被请求的保护级别设置成正确值。然后，IA-32 硬件使用这个被请求的保护级别去阻止任何非法行为：如果某一个段具有比被请求的保护级别更高的保护级别，则该段不能被使用这些参数的系统进程访问。

页式虚拟存储器示例：64 位的 Opteron 存储器管理

AMD 的工程师发现以上描述的保护模型几乎很少使用。主流模型是由 80386 提出的平面 32 位地址空间，它将所有的基址段寄存器的值置为 0。因此，AMD 在 64 位模式下分配了多个段。假设段基址为 0 并忽略范围字段。页面大小为 4 KB，2 MB 和 4 MB。

尽管实现起来可以用更少的位来简化硬件，但 AMD64 系统结构的 64 位虚拟地址映射到了 52 位物理地址上。例如 Opteron 使用 48 位虚拟地址和 40 位物理地址。AMD64 需要高 16 位的虚拟地址来标记扩展低 48 位，这称为规范格式。

页表大小对于 64 位地址空间而言也还是太大了。因此，AMD64 采用多级分层页表结构映射地址空间以使页表大小合适。分级的级数取决于虚拟地址空间的大小。图 C.26 显示了 Opteron 的 48 位虚拟地址的 4 级转换。

每个这样的页表的偏移量来自 4 个 9 位的域。地址转换开始时，在第四级基址寄存器加上页映射的初始偏移量，然后在此读存储器得到下一级页表的基址。下一级地址偏移量依次加上新的地址，并再次访问可得到第三级页表的基址。如此反复进行。最后的地址加上最终得到的基址，使用此地址和读存储器得到页面相关的物理地址。此地址加上 12 位页偏移量得到完整的物理地址。注意，Opteron 系统结构的页表也适合单独的 4 KB 页面。

Opteron 在每个页表内采用 64 位的页表项。前 12 位为以后使用预留，另外 52 位包含物理页号，最后 12 位包括保护和使用权信息字段。尽管此字段在每级页表中都不同，以下是基本的字段：

- 当前位：说明页面当前在存储器中。
- 读/写位：说明页面是否只读或只写。
- 用户/管理位：说明用户是否能访问此页或由上面三个特权级所限制。
- 重写位：说明页面是否被修改。
- 访问位：说明此位被清除前，页面是否在读或写。
- 页面大小：说明最后一级页面是 4 KB 还是 4 MB；如果是 4 MB，则 Opteron 仅仅使用三级页面而非四级。
- 非执行位：80386 保护机制中没有此位，非执行位用来阻止代码从某些页中执行。
- 页级 Cache 使能：说明页面能否被缓存。
- 页级写直达：说明是否允许对数据 Cache 写回或写直达。

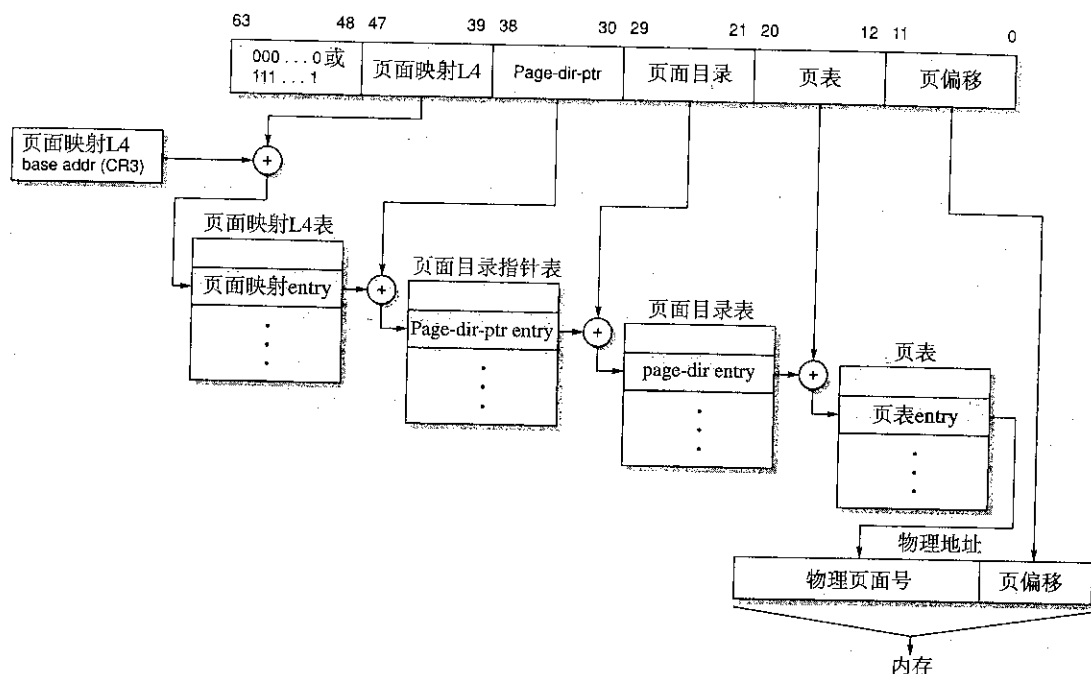


图 C.26 Opteron 虚拟地址映射。Opteron 的虚拟存储器系统中采用了 4 级页表结构，支持 40 位的物理地址。每个页表有 512 个项，因此每级页宽为 9 位。AMD64 系统结构的文档允许虚拟地址大小由当前的 48 位增加至 64 位，物理地址大小由当前的 40 位增加至 52 位。

由于 Opteron 通常在一次 TLB 缺失中会通过四级页表，故有三个潜在的检查保护约束的位置。Opteron 仅遵从下级的 PTE 并检查其他级的有效位是否设置。

由于 PTE 为 8 字节长，每个页表有 512 个项，Opteron 有 4 KB 大小的页面，页表正好是一页长。每级页宽为 9 位，页偏移量为 12 位。这种方法为标记扩展预留了 $64 - (4 \times 9 + 12) = 16$ 位的空间，以确保地址的规范化。

虽然我们已经解释了合法地址的转换，但怎样阻止用户进行非法的地址转换，以及如何避免由此产生的失效呢？页表自身的保护避免了用户程序对其进行写操作，因此，用户可以使用任何虚拟地址，但是通过控制页表项，操作系统能够控制哪些物理地址可以被访问。多个进程共享存储器是通过使各自地址空间中的一个页表存储字指向同一物理存储器页来实现的。

Opteron 使用 4 个 TLB 以减少地址转换时间，2 个用于指令访问，2 个用于数据访问。和多级 Cache 类似，Opteron 通过 2 个更大的二级 TLB 来减少 TLB 缺失：1 个用于指令访问，1 个用于数据访问。图 C.27 给出了每个 TLB 的关键参数。

参数	描述
块大小	1 PTE (8 字节)
L1 命中时间	1 个时钟周期
L2 命中时间	7 个时钟周期
L1 TLB 大小	指令和数据 TLB 都是 40 个 PTE，每个 PTE 映射 32 个 4 KB 页面或 8 个 2 MB 或 4 MB 页面
L2 TLB 大小	指令和数据 TLB 都是 512 个 PTE，4 KB 页面
块选择	LRU
写策略	(未应用)
L1 块放置策略	全相联
L2 块放置策略	4 路组相联

图 C.27 Opteron 中一级和二级指令、数据 TLB 的存储层次参数

小结：32 位 Intel Pentium 中的保护和 64 位 AMD Opetron 保护的对比

Opteron 的存储管理是如今大部分桌面和服务器的典型方法，依靠页级地址变换和操作系统的纠错操作，Opteron 的存储管理可以提供安全的多进程共享。Intel 也提出了一些可作为替代选择的方案。但实际上，这些方法沿袭了 AMD64 系统结构所倡导的设计风格。因此 AMD 和 Intel 都支持 80x86 的 64 位扩展。由于兼容性的原因，他们都支持分段保护机制。

分段保护模式看起来比 AMD64 保护模式更难于建立，事实的确如此。由于只有少量的用户使用这种严密的保护机制，而且这种保护模式与 UNIX 简单页式保护并不匹配，因此，它仅能被专门为这类计算机设计操作系统的人所使用。

C.6 谬误和易犯的错误

即使在存储器层次结构的回顾中，也有谬误和易犯的错误。

易犯的错误：地址空间大小。

仅仅在 DEC 和 Carnegie Mellon 大学研制出 PDP-11 系列计算机后 5 年，它的致命缺点就渐渐显露了出来。IBM 在 PDP-11 诞生的 6 年前发布的一种系统结构在 25 年后仍然流行，而且只做了少量的改动。在 PDP-11 停产后，DEC 研制出了 VAX，虽然备受批评（被认为在其中加了些不必要的功能），但还是卖出了几百万台。为什么会这样呢？

PDP-11 的致命缺陷是由于它的地址空间只有 16 位，而 IBM 360 有 24 到 31 位，VAX 有 32 位。地址位的多少限制了程序的大小，所以程序大小和需要数据的大小之和不能超过 2 的地址位数次幂。地址长度很难改变的原因是因为它决定了能包含地址的任何东西的最小宽度：PC、寄存器、存储字以及有效地址算法。如果设计时没有想到扩充地址长度，那么以后想要再扩展的可能性很小，除非重新设计新的系列机。Bell 和 Strecker[1976]是这样说明这个问题的：

在计算机设计中，只有一个错误是无法挽回的——没有足够的地址位用来寻址和进行存储器管理。PDP-11 和大多数著名的计算机都犯了同样的错误。[p.2]

曾经取得成功但最终因为地址位不够长而被淘汰的机型包括 PDP-8, PDP-10, PDP-11, Intel 8080, Intel 8086, Intel 80186, Intel 80286, Motorola 6800, AMI 6502, Zilog Z80, CRAY-1 和 CRAY X-MP。

即使是老牌的 80x86 系列也面临扩充地址空间的问题, 在 1985 年首先是 Intel 80386 扩充到 32 位, 现在又随着 AMD Opteron 一起扩充至 64 位。

易犯的错误: 考虑存储器层次结构的性能时忽略操作系统的影响。

图 C.28 给出了在运行三大任务时, 由于操作系统原因造成的存储器停顿时间。大约有 25% 的停顿时间与操作系统有关: 除了操作系统自身的缺失外, 还有与操作系统交互的应用程序缺失。

工作 负载	时间								
	缺失		程序缺失造成的时间损失		操作系统缺失造成的时间损失				
	应用 程序 缺失	操作 系统 缺失	应用 程序 内部 缺失	程序和 操作系 统冲突 造成的 缺失	操作 系统 指令 的缺失	移植 带来的 数据 缺失	块操 作中的 数据 缺失	其他 操作 系统 缺失	操作系统 缺失和程 序冲突造 成的时 间损失
Pmake	47%	53%	14.1%	4.8%	10.9%	1.0%	6.2%	2.9%	25.8%
Multipgm	53%	47%	21.6%	3.4%	9.2%	4.2%	4.7%	3.4%	24.9%
Oracle	73%	27%	25.7%	10.2%	10.6%	2.6%	0.6%	2.8%	26.8%

图 C.28 操作系统和应用程序的缺失次数和损失时间。由于操作系统的开销, 应用程序的运行时间一般会增加 25%。每个处理器有一个 64 KB 的一级指令 Cache, 一个 64 KB 的一级数据 Cache 和一个 256 KB 的二级数据 Cache; 所有的 Cache 都是直接映射, 块大小是 16 字节。这些数据在 Silicon Graphics POWER 工作站 4D/340 上得到, 这款机器有 4 个 33 MHz 的 R3000 CPU, 在 UNIX System V 下运行 3 个不同的任务; 第一个是 Pmake, 一个同时编译 56 个文件的并行编译器; 第二个是 Multipgm, 一个和 Pmake 配合的并行数值程序, 并有 5 个窗口会话; 第三个是 Oracle, 使用 Oracle 数据库运行一个受限的 TP-1 测试 (数据由 Torrellas, Gupta 和 Hennessy[1992]整理)

易犯的错误: 由操作系统改变页面大小。

Alpha 设计者计划通过增加页面大小, 甚至使之达到虚拟地址的大小, 来改进 Alpha 的系统结构。当后续版本的 Alpha 需要增加页面大小时, 操作系统的设计者必须相应地调整虚拟存储系统使之能处理更大的地址空间, 而不是一直保持 8 KB 的页面。

TLB 的高缺失率已经引起了其他计算机系统结构设计者的关注, 因此, 他们使 TLB 可以支持多个更大的页。这是为了在必要时, 程序员可以分配一个足够大小的页, 从而节省 TLB 项。经过 20 多年的探索, 大部分操作系统都为一些特殊的功能采用了“超页”技术: 比如映射显存或其他 I/O 设备, 或者为数据库代码分配一个非常大的页。

C.7 结论

要构建一个与飞速发展的处理器相匹配的存储系统, 其难度是很大的。尤其是内存所用的原材料在任何计算机里都是一样的, 没有其他的选择。局部性原理从磁盘到 TLB 级证明了当前计算机系统中存储层次结构的稳定性。

但是在2006年，不断增加的相关存储时延，消耗了大量时钟周期。对程序员和编译器设计者而言，这意味着如果要确保其程序更好地运行，就需充分利用 Cache 和 TLB 的参数。

C.8 历史回顾和参考文献

在随书光盘的 K.6 节中，我们分析了 Cache、虚拟存储器和虚拟机的发展史。在这些技术的发展过程中，IBM 扮演了极其重要的角色。其中也包含了进一步阅读的参考文献。

关于光盘

随本书附带的光盘内容包括：

- 附加参考文献。这些附加的参考文献涉及到了大量的主题，包括专用体系结构、嵌入式系统和专用处理器等。
- 历史回顾和参考文献。附录 K 中的几节探讨了本书中一些章中出现的关键概念，还提供了供读者深入阅读的参考文献。
- 搜索引擎。包含了一个搜索引擎，这使得搜索课本及光盘中附录的内容变得可能。

光盘上的附录

- 附录 D：嵌入式系统
- 附录 E：互连网络
- 附录 F：向量处理器
- 附录 G：VLIW 和 EPIC 的硬件与软件
- 附录 H：大规模多处理器和科学应用
- 附录 I：计算机算术
- 附录 J：指令系统结构概述
- 附录 K：历史回顾与参考文献

计算机系统结构

—— 量化研究方法 (第四版)

John L. Hennessy David A. Patterson

多处理器时代的到来已经势不可挡。当我们告别单核处理器时代进入芯片多处理时代之际, Hennessy 和 Patterson 著作最新版本的推出无疑是逢时之作。很少有其他著作能像本书一样对计算机系统结构产生过如此巨大的影响, 而这一最新版本必定在未来很长一段时间内将是这一领域的权威之作。

—— Luiz Andre Barroso, Google 公司

如果你问下列哪个可算做经典: 甲壳虫乐队, HP 计算器, 巧克力曲奇饼, 还是这本书? 我会告诉你, 它们都很经典, 因为它们都经受住了时间的考验!

—— Robert P. Colwell, Intel 首席架构师

这是一本系统介绍计算机系统的设计基础、指令集系统结构、流水线和指令级并行技术、层次化存储系统与存储设备、互连网络以及多处理器系统等重要内容的畅销书。在这一最新版本中, 作者更新了与单核处理器到多核处理器历史发展过程相关的内容。同时使用了广受好评的“量化研究方法”进行计算设计, 并阐述了多种可以实现并行的技术, 这些技术恰恰是展现多处理器系统结构威力的关键。在介绍多处理器时, 作者不仅讲述了处理器的性能, 而且还介绍了处理器性能之外的其他设计要素, 包括功耗、可靠性、可用性和可信性等。

自上个世纪以来, 在成本和性能等方面的提高需要不断地创新——这些创新主要来自这本经典著作所涵盖的基础技术中。

本书主要特点

- 每章中的“综合”小节主要关注业界的各种最新技术, 包括 Sun Niagara 处理器, AMD Opteron 处理器以及 Intel Pentium 4 处理器。
- “回顾”部分包括了与正文内容密切相关的基本和中级原理。
- 随书附带光盘中的“参考文献”收录了一些特邀学术专家的文章, 其中包括嵌入式系统、向量处理器、互连网络和大规模多处理器系统等很多方面的内容。
- 每章最后都有一个由工业界或学术界的专家提供的“范例分析”, 以及与之配套的习题, 以便读者更加深入地理解和掌握每章中所论述的关键概念。



责任编辑: 谭海平

责任美编: 陈晓磊

本书贴有激光防伪标志, 凡没有防伪标志者, 属盗版图书。

本书译自原版 Computer Architecture: A Quantitative Approach, Fourth Edition, 并由 Elsevier 授权出版。



ISBN 978-7-121-04769-5



9 787121 047695 >

定价: 69.80 元
(附光盘一张)